

REPORT DOCUMENTATION PAGE

AD-A201 921

ECTE

/ 23 1988

JLE

1b. RESTRICTIVE MARKINGS

3. DISTRIBUTION/AVAILABILITY OF REPORT

Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

ARO 23714.7-MA-H

6a. NAME OF PERFORMING ORGANIZATION

Howard University

6b. OFFICE SYMBOL
(If applicable)

7a. NAME OF MONITORING ORGANIZATION

U. S. Army Research Office

6c. ADDRESS (City, State, and ZIP Code)

Washington, DC 20059

7b. ADDRESS (City, State, and ZIP Code)

P. O. Box 12211
Research Triangle Park, NC 27709-22118a. NAME OF FUNDING/SPONSORING
ORGANIZATION

U. S. Army Research Office

8b. OFFICE SYMBOL
(If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

DAAL03-86-6-0085

8c. ADDRESS (City, State, and ZIP Code)

P. O. Box 12211
Research Triangle Park, NC 27709-2211

10. SOURCE OF FUNDING NUMBERS

PROGRAM
ELEMENT NO.PROJECT
NO.TASK
NO.WORK UNIT
ACCESSION NO.

11. TITLE (Include Security Classification)

Analysis of Blending Algorithms in Computer Graphics (Unclassified)

12. PERSONAL AUTHOR(S)

Dr. Ronald J. Leach

13a. TYPE OF REPORT
Final13b. TIME COVERED
FROM 8/1/86 to 7/31/8814. DATE OF REPORT (Year, Month, Day)
October 17, 1988

15. PAGE COUNT

16. SUPPLEMENTARY NOTATION

The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES

FIELD

GROUP

SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

computer graphics, solid modeling, blending surface,
minimal

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The research on this project was directed towards determination of fast algorithms which produce surfaces which blend together other surfaces and for which the blending surface has geometric significance. Many display algorithms were analyzed for numerous architectures; the best algorithms were determined. Blending surfaces which incorporated geometric information such as minimizing surface area were studied. New degrees of freedom in the evaluation of certain surfaces were discovered.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT

☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL

Ronald J. Leach

22b. TELEPHONE (Include Area Code)

(202) 636-6650

22c. OFFICE SYMBOL

ANALYSIS OF BLENDING ALGORITHMS
IN COMPUTER GRAPHICS

FINAL REPORT

OCTOBER 19, 1988

U.S. ARMY RESEARCH OFFICE

DAAL 03-86-G-0085

HOWARD UNIVERSITY

PROJECT DESCRIPTION

1. INTRODUCTION

(fast)

The research on this project centered around the analysis, development, and implementation of algorithms for representing a surface which blends together two or more intersecting surfaces. The blending surface should provide a smoother transition than is available when simply considering the intersection of the surfaces. The algorithms developed must be efficient because the major portion of computing time in a solid modeling system should be devoted to the important problems of data representation, object display, surface analysis (ray tracing, shadowing, etc.) and scene analysis (hidden line/surface removal, etc.) The important features of a blending surface are speed of computation, ability to be incorporated into the other features of a solid modeling system (such as hidden line/surface removal), and the visual quality of the blended image.

This report is organized as follows. Section 1 is an introduction. Section 2 briefly describes some basic concepts from solid modeling. Section 3 is a description of the organization of the research, major findings, and publications. Section 4 describes additional directions for research.

Keywords: applied geometry, degree of freedom (KR)



Accession For	
NTIS CPA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special
A-1	

2. SOME CONCEPTS FROM SOLID MODELING

Computer aided design (CAD) systems depend upon representations of solid objects which use an abstract, geometrical model rather than simply representing them as collections of pixels displayed on a CRT screen. There are several common methods for modeling geometric information, each with its own advantages and disadvantages. The most common methods are the boundary representation in which an object is described by its boundary; constructive solid geometry, in which an object is described by its boundary; constructive solid geometry, in which an object is described by an algorithm for constructing it from geometric primitives such as spheres or boxes; and a data-structured oriented method such as octrees, in which an object is described by the portions of space it occupies.

Each of these methods admits additional refinements. For example, the boundary representation of a single surface may be given by the implicit representation $F(x,y,z)=0$, the explicit representation of the form $z=f(x,y)$ which is obtained by solving the implicit equation (where possible and where a unique solution exists), and the explicit parametric representation where x , y , and z are described by a set of parametric equations.

The implicit representation is convenient for finding the intersection of two surfaces. However, the other methods are often more convenient for actually graphing objects.

Research during the project period has centered on the boundary representation method using all three of the techniques of implicit and explicit surface description.

3. SUMMARY OF RESEARCH FINDINGS

The research on this problem has taken two directions: efficiency of computational algorithms and development/implementation of mathematical models.

The research on the efficiency of computational algorithms has reinforced a well-known phenomenon in computer graphics - that very high speed cpu's, parallel/distributed processing, or specialized graphics hardware are frequently necessary for responsive systems. In the paper "Evaluating the Performance of a User Interface" (Computes and Graphics volume 11 no. 2 (1987), 141-146), algorithms for evaluating performance of display systems (especially window display and menu selection systems) were given. These algorithms are appropriate for general systems, with no particular emphasis on solid modeling systems.

The primary focus of "Complexity of Computer Algorithms" (Rocky Mountain J. Math volume 17 (1987), 167-187) was general computer algorithms. However, particular algorithms for polynomial evaluation via look-up tables and Tchebycheff polynomials were also presented, polynomial evaluation algorithms are necessary for any method of graphical representation using bicubic or similar patches. This work has naturally lead to the study of special purpose parallel algorithms for fast polynomial evaluation:

Publications in the area of parallel algorithms relevant to computer graphics include "Ada Software Metric and Their Limitations" (Proc. Joint Ada Conference, Washington, DC. (March 1987), 285-293) in which formal measurements of software complexity were made, "Use of Concurrent Tasking Paradigms for the Design of Ada Programs" (Proc 6th Annual Conference on Ada Technology, Washington, D.C. (March 1988), 153-156) in which formal models of concurrent/parallel programs were used, and "Actual Complexity of Parallel Evaluation of Low Degree Polynomials" (being reviewed for publication) in which several algorithms are evaluated. The first two papers mentioned in this paragraph consider high-level algorithms while the third paper is concerned with low level computations including counts of memory accesses and inter-process communications. An important result in that paper is that "efficiency" larger than 1 is possible for evaluation of cubic polynomials at multiple points. The efficiency is defined here as

$$N \cdot T(N) / T(1)$$

where $T(1)$ represents the time if one processor is used and $T(N)$ represents the time if N processors are used. This high efficiency is based on a pipelined architecture based on deCastlejan's algorithm.

Comparisons between this specialized architecture and several algorithms such as Knuth's, Horner's and a finite difference method are also given; none of the sequential, non-pipelined algorithms achieve efficiency of 1.

An offshot of some of this work was two papers primarily on education, "Experiences Teaching Concurrency in Ada" (AdaLetters vol. 7 no. 2 (1987), 40-41) and "A Suggested Topic for the First Course in Computer Science" (SIGCSE Bulletin, vol 30, no. 2 (1988), 40-43).

The other research direction that was considered involved the development/implementation of mathematical algorithms for blending surfaces. The goal of this research was to allow the inclusion of geometric information into blending surfaces. the initial geometric information used was the minimization of the surface area of the blending surface.

The first technique used was based on the observation that any such blending surface which minimizes surface area must be a minimal surface. Minimal surfaces are surfaces which satisfy the non-linear partial differential equation

$$z_{xx}(1+z_y^2) - z_x z_y z_{xy} + z_{yy}(1+z_x^2) = 0$$

(assuming z is a function of x and y). For computational purposes, the most appealing minimal surfaces for blending purposes are those of low degree. Therefore the first objective in this area of research was to obtain minimal surfaces of low degree.

Surfaces of degree 3 or 4 have been classified by Salmon as falling into one of several classifications. The determination of which of these low degree surfaces, if any, is a minimal surface is a formidable computational problem. After applying various simplifying transformations to the equations of the surfaces, the equations were used as input to the symbolic manipulation package MACSYMA for determining if the surface satisfied the minimal surface equation. Results were obtained for a few of the possible categories of surfaces; however, problems with the disk drives of the computer prevented a complete solution of the problem. (The computer hardware problems have been documented in the most recent interim project report. The disk drive problems generally made the use of large virtual memory space impossible and thus only incomplete results could be obtained because of the large intermediate size of algebraic expressions given by MACSYMA before simplification).

Since only incomplete results were obtained in the search for low degree minimal surfaces, with the implicit representation, the project now considered the use of explicit parametric representation. A classical result of Weierstrass was used as the starting point - parametrizations of minimal surfaces arise from integrating from 0 to z the functions

$$\begin{aligned} (1-g^2) w/2 \\ (1+g^2) w/2i \end{aligned}$$

gw

where g and w are analytic functions on some domain. The parameters are the real and imaginary parts of the independent variable z . An important contribution here was the observation that these analytic functions have additional degrees of freedom which provide 6 additional degrees of freedom in the simplest known surfaces (the Enneper's surface) and more for surfaces of higher algebraic degree. This research is incorporated in the papers "Minimal Blending Surfaces" and "Geometric Considerations in Blending Surfaces". Both of these papers have been submitted for publication and are still in the reviewing process. Each of these papers indicate the use of these minimal surfaces as blending surfaces.

The papers have concentrated on simple problems in blending surfaces which are sections of minimal surfaces. The additional degrees of freedom mentioned above are used for curve fitting and for visual appeal.

4. FUTURE DIRECTIONS

Current work is directed towards using other geometrical constraints such as minimizing additional volume due to blending surfaces (with some tangency information added) and towards using methods of the calculus of variations for problems posed in terms of surfaces which are described parametrically. The fundamental idea is to use a Ritz method to find good approximate solutions which are in the class of low degree surfaces. It is expected that results will be submitted for publication by the end of 1988.

Additional work on the determination which of the surfaces of degree 3 or 4 as minimal surfaces will be postponed until sufficient reliable disk drive capacity becomes available.

APPENDIX 1

Students Supported

- | | | |
|-------------------|------------------------|-----------|
| 1. Sileshi Kassa | MS Computer Science | May 1987 |
| 2. Darlene Bond | MS Computer Science | May 1988 |
| 3. Michael Atogi | MS Computer Science | June 1988 |
| 4. Robert Bennett | no graduate degree yet | |

APPENDIX 2

Final Budget

DATE RUN 08/01/88 ** H O W A R D U N I V E R S I T Y ** REPORT PAGE 9703
 TIME RUN 21:17:49 FINANCIAL RECORDS SYSTEM PROGRAM ID FBM082
 FBM080 - H1 ACCOUNT STATEMENT IN WHOLE DOLLARS FOR 07/29/88 ACCOUNT PAGE 1
 DIVISION-DEPT = 29 00035

ACCT: 5-24740 ANALYSIS OF BLENDING ALGORITHMS TO: LEACH, DR. RONALD J
 DEPT: 00035 01 ENGR.

SUB CODE DESCRIPTION	BUDGETS		ACTUAL		OPEN COMMITMENTS	BALANCE AVAILABLE	PERC USED
	ORIGINAL	REVISED	CURRENT MONTH	FISCAL YEAR			
1000 SALARY-ADMIN	17,908	1,832				1,832	0
1010 FAC SAL - PROFESSORS		17,889	4,884	4,884	699		100
1200	7,300						0
1230 SALARY-GRAD. ASSIST	7,300	4,270			4,269	1	100
SALARIES, WAGES, OTHER	32,508	23,771	4,884	4,884	699	1,833	93
1900 STAFF BEN @ 26% S&W	8,452	814				814	0
1910 EMPLOYEE BENEFITS		5,574	1,213	1,213	5,574		100
EMPLOYEE BENEFITS	8,452	6,188	1,213	1,213	5,574	814	90
2000 SUPPLIES & EXPENSES	3,000	1				1	0
2010 OFFICE SUPPLIES		109			109		100
2040 COPYING/DUPLICATING		500			500		100
2220 SOFTWARE		350			150		100
2890 MINOR FURN & EQUIPT		148			148		100
3210 BOOKS, PER, FLM-NONL		41			41		100
3830 BOOKSTORE		200			200		100
4000 TRAVEL	3,000	73				73	0
4020 TRAVEL-OTHER DOMESTI		2,795	1,139	1,139	2,795		100
4200 TRAINEE COSTS	3,000						0
4210 TRAINEE COSTS-STIPEN		20,134			19,800	334	98
4240 TRNEE CST-TUITN/FEEs		3,000			3,000		100
SUPPLIES & EXPENSES	9,000	27,348	1,139	1,139	26,741	407	99
5000 EQUIPMENT	9,320	627				627	0
5410 RESRCH/SCIENTIF EQUI		7,401			7,401		100
EQUIPMENT	9,320	8,028			7,401	827	92
8000 TUITION	3,000	1				1	0
8010 SCHOLARSHIPS		8,452			8,452		100
SCHOLARSHIPS/FELLOWS	3,000	8,452			8,452	1	100
9000 IND. COST @ 90% MTDC	42,284	30,756	607	607	30,756		100
TOTAL EXPENSES	104,544	104,544	7,622	7,622	100,363	3,282	97
ACCOUNT TOTAL	104,544	104,544	7,622	7,622	100,363	3,282	97

DATE RUN 08/01/88
TIME RUN 21:17:49
FBM090 - H1

** H O W A R D U N I V E R S I T Y **
FINANCIAL RECORDS SYSTEM
ACCOUNT STATEMENT IN WHOLE DOLLARS FOR 07/29/88
DIVISION-DEPT = 29 00035

REPORT PAGE 9704
PROGRAM ID FBM092
ACCOUNT PAGE 2

ACCT: 5-24740
DEPT: 00035

ANALYSIS OF BLENDING ALGORITHMS

TO: LEACH, DR. RONALD J
01 ENGR.

SUB CODE DESCRIPTION	BUDGETS		ACTUAL		PROJECT YEAR	OPEN COMMITMENTS	BALANCE AVAILABLE	PERC USED
	ORIGINAL	REVISED	CURRENT MONTH	FISCAL YEAR				

THIS IS A SUMMARY OF YOUR ACCOUNT SUPPORTING OUR MONTH TRANSACTIONS
THE DETAIL IS ON FBM091. PLEASE CALL EXT 8505 WITH QUESTIONS.

OPEN COMMITMENTS STATUS

ACCOUNT	REF. NO.	DATE	DESCRIPTION	ORIGINAL AMOUNT	LIQUIDATING EXPENDITURES	ADJUST- MENTS	CURRENT AMOUNT
5-24740-2220 U	P138438	03/18	SYMBOLIC COMPUTATION	200.00			200.00
5-24740-1010 U	S000001	05/13	LEACH, RONALD J	10,028.27	9,327.04		699.23
5-24740-4020 U	T118204	08/07	RONALD J. LEACH	1,094.33	1,139.20	44.87-	COMPLETED
** ACCOUNT TOTAL **				11,320.60	10,466.24	44.87-	899.23

FOR QUESTIONS ON PURCHASE ORDERS, CALL EXT 6081 (SUPP.), 5794 (SERV.)
AND 5534 (GRANT). FOR OTHER COMMITMENTS CALL EXTENSION 8507 OR 8512

DATE RUN 08/01/88 ** H O W A R D U N I V E R S I T Y ** REPORT PAGE 9705
TIME RUN 21:17:49 FINANCIAL RECORDS SYSTEM PROGRAM ID FBM092
FBM091 REPORT OF TRANSACTIONS FOR 07/29/88 ACCOUNT PAGE 1
DIVISION-DEPT = 29 00035

ACCT: 5-24740 ANALYSIS OF BLENDING ALGORITHMS TO: LEACH, DR. RONALD J
DEPT: 00035 01 ENGR.

SUB CODE	DESCRIPTION	DATE	EC	REF.	2ND REF.	J.E. OFFSET ACCOUNT	BUDGET ENTRIES	CURRENT REV/EXP	COMMITMENTS	BATCH REF.	DATE
1010U	LEACH, RONALD J	07/15	082	S000001		0-98888-1810CR U		2,331.78	2,331.78-	PAYC46	880715
	U LEACH, RONALD J	07/29	082	S000001		0-98888-1810CR U		2,331.78	2,331.78-	PAYC49	880728
1010	CM TOTAL FAC SAL - PROFESSORS							4,663.52	4,663.52-		
1910U	FB 0988	07/29	081	0000952	0000225	2-90000-1920CR U		1,212.52		PROJ95	880728
1910	CM TOTAL EMPLOYEE BENEFITS							1,212.52			
4020	RONALD J*LEACH	07/14	048	T116204	400245			1,139.20	1,139.20-	APC988	880714
	U RONALD LEACH	07/07	053	T116204					44.87	EP8018	880707
4020	CM TOTAL TRAVEL-OTHER DOMESTI							1,139.20	1,094.33-		
9000U	-IC0288	07/29	081	0000953	0000091	1-40018-0800CR U		807.00		ICCU95	880728
9000	CM TOTAL IND.COST @ 90% MTDC							807.00			

*** ACCOUNT TOTAL *** 7,622.24 5,757.85-

APPENDIX 3

Publications

ADA SOFTWARE METRICS AND THEIR LIMITATIONS

Ronald J. Leach

Howard University

ABSTRACT

A major goal of software engineering research is the development of metrics which measure the complexity and maintainability of programs, with a small portion of this effort directed specifically towards programs written in Ada. This paper will focus on two main themes. The first theme will be the development of metrics that specifically reflect the complexity of programs in Ada. The second theme will be an investigation of the theoretical limits of metrics as measures of program complexity in general.

1. INTRODUCTION

There has been a considerable amount of research activity directed towards the development of measurements of complexity of programs which have high correlation with programming effort. See for example references [1], [4], [5], [6], [7], [8], [9], [11], [13]. Much of the work may be broadly classified into three categories, each related to a model of programming complexity.

The first model assumes that the programming complexity is the sum of the programming complexities of the various modules making up the program. A typical example of this research [4] uses measures such as the number of operators and operands, number of distinct operators and operands, etc. No special use is made of control structures; for example, a GOTO statement is treated as having an operator and operand. Another example of this type is the cyclomatic measure of McCabe [8]. Here the branching and flow of control is considered, with the major concern being computation of

a number called the cyclomatic complexity which is basically Euler's formula applied to a graph which represents the program. Kafura and Henry [5] call these metrics microlevel.

The second model assumes that the particular modules comprising a program are relatively straightforward and that the major factors in program complexity are the interconnections between these modules. A primary example of this type of research is [5] and the references indicated there.

The third model [3] assumes that a primary factor in program complexity is the experience of the programmer with other factors such as the goals of program efficiency or storage constraints having some effect on the complexity.

Very few researchers consider more than one of these models; Kearney, et al [6] is an exception.

We note that much of the literature on software metrics is concerned with the coding phase of software development. Few articles explicitly address the points made by Carrio [3], Ramamoorthy [12] and many others that maintenance is a major portion of the software life cycle. Carrio states that many of the changes are caused by what he calls "pseudo-maintenance" activities which change the scope of the project by adding features or changing requirement specifications. Ramamoorthy describes a sample of 282 programs with failures in which 38.2% are caused by problems in the requirements/specification level.

One of the claims made for Ada is that it will reduce the total amount of complexity of software in a particular installation by encouraging reusable code and to some degree by acting at least in part as a program design language. In the conclusion of this paper we will make some observations about these claims.

In this paper we consider a variety of software metrics which are applied to over 30 programs written in the language Ada. The paper is organized as follows. In section 2 we describe the results obtained by applying several metrics such as in [1],[4], [5], [7], [8]. In section 3 we consider the same data using some new metrics.

Sections 4 and 5 are concerned with limitations of software metrics for Ada programs. Section 4 introduces the concept of a time-varying metric. Such metrics have the goal of measuring that portion of a program most likely to change because of a change in program specifications. The metrics are also evaluated on the same data set. Relevance of such metrics and the associated statistics to the software life cycle is discussed.

Section 5 of the paper includes a discussion of inherent complexity of programs and the resulting limitations of software metrics.

Section 6 provides a summary of the results obtained and some suggestions for future work.

2. APPLICATION OF EXISTING METRICS

This paper has the twin goals of developing software metrics for Ada programs and describing the limitations of such metrics. In this section we discuss the development of metrics.

When developing new metrics, it is instructive to examine the behavior of some of the classical metrics on a set of sample programs. The sample programs were selected at random from a variety of textbooks on Ada; they range in length from 33 to 236 lines of code. We emphasize that this is not a formal experiment. Instead, we consider the examples given here as providing experience in the collection of data for the development of more complete metrics such as those considered in the next section.

With these caveats in mind we present the data in Table 1 for these programs. We use the notion of operator and operand as described in [4]. The correlation between the Halstead and McCabe metrics for these programs is a low 0.139 explaining only 37% of the variance. A graphical view of these metrics is given in figure 1 at the end of the paper where they are compared with a new Ada metric.

Table 1

Program	Halstead	McCabe	Lines
1	7611	5	33
2	5148	6	51
3	13485	4	55
4	215985	8	111
5	46189	8	115
6	42169	5	79
7	32733	4	42
8	73171	7	72
9	13537	14	91
10	35255	7	74
11	19162	8	58
12	12832	2	45
13	5923	3	39
14	16476	5	67
15	47071	2	81
16	37405	5	50
17	2482	3	36
18	19826	4	68
19	2145	3	41
20	1572	3	65
21	3381	1	32
22	10806	3	39
23	160794	3	91
24	100991	12	92
25	47611	6	72
26	44781	7	66
27	14136	4	68
28	6441	4	48
29	64893	24	236
30	5159	3	50

3. ADA-SPECIFIC METRICS

In the previous section we saw that for a large collection of Ada programs, there is low correlation between the frequently used Halstead and McCabe metrics. Clearly neither metric completely measures software complexity. Thus we need to examine metrics providing "orthogonal" views of a program. Such metrics will use some of the ideas of the interconnection ideas of [5] as well as [7].

We consider a collection of Ada language features whose presence may explain the wide variation between the various metrics. These features are grouped by their perceived effects on language level, programmers' specific abilities, portability, and verifiability. We consider only those features that are specific to Ada and not available in other languages such as Pascal. The reason for this is that an Ada program written only using Pascal-like features can have its software quality measured by obvious

translations of Pascal metrics (assuming that there are adequate metrics for Pascal programs).

A. FEATURES DUE TO LANGUAGE LEVEL.

1. Name Equivalence

For example, the declarations
A,B: INTEGER;
C: INTEGER;
Type DATATYPE is new INTEGER;
D: DATATYPE;
Type INT_TYPE is new INTEGER;
E: INT_TYPE;

allow A,B, or C to be considered the same type, but D is considered a different type. This has no effect on McCabe's metrics. There is no effect on Halstead's if we consider only executable statements and increase the number of lines of code by 4.

The variables A,B, and C are all declared as being of type INTEGER. Writing the declaration in this form require more source code text than does

A,B,C: INTEGER

However if it is accompanied by a comment explaining the significance of the variable C, then the effect of this longer form of the declaration is to slightly increase readability of the code and hopefully to slightly reduce program complexity.

The declaration of D as being a distinct type (called a derived type in Ada) allows storage of D and allows many operations to be performed on D. However, it does not allow operations such as the addition of a variable to type INT_TYPE to a variable of type DATATYPE. This actually reduces the complexity of the program since it precludes "accidental" errors such as adding a Zip Code to a Social Security number and expecting a sensible result.

Note that this phenomenon does not occur in a language with only "structure equivalence" of names. In such languages, addition of A and D is a legitimate operation.

2. Generics

Generic packages provide an opportunity for data abstraction. As such, they represent an opportunity for the program to represent an algorithm more clearly. Thus the effect of "generics" will be to reduce the complexity of the software during the design and coding phase since among other things they reduce the number of subroutines and lines of code of the program. However, as was pointed out in [1], generics form a template whose

correctness in a particular program is difficult to test without exhaustive consideration of all cases of instantiations.

B. FEATURES WHICH INFLUENCE PROGRAMMER UNDERSTANDING.

1. Tasking

The ability to specify operations which need not be executed sequentially is one of the major features of Ada. The writing and debugging of programs involving tasks is complicated by the fact that some errors will become known only when certain orders of statement execution are followed and these particular orders often occur long after the testing phase.

Tasking is also one of the few factors present in all phases of the program life cycle from specification to maintenance. Therefore it must be included in metrics which are to be applied at various times during the life cycle. In addition, tasking used to improve performance by splitting execution of processes onto many processors should probably change when the number of processors available in any implementation of the software increases. This facet of tasking therefore will also affect the portability of code somewhere during the life cycle.

Because of the complexity introduced by tasking, we must treat arrays of tasks separately from the way we treat arrays of data objects. Declaration of an array of data objects does not change any measurement of software based only on executable statements. An array of tasks provides far more opportunity for errors in interconnection between two tasks than do one or two tasks. Thus the number of elements in an array of tasks has a great affect on any reasonable Ada software metric.

An even worse situation is caused by the Ada language allowing the creation of pointers to tasks. Each additional task increases software complexity. However, the number of tasks cannot be determined until after execution of the program. We return to this point later.

2. Subfeatures of tasking

Many of the problems occurring in software which allows concurrency are caused by synchronization of processes. With this in mind we observe that the reserved words "select", "accept", "entry", "delay",

"abort" in the context of tasks must increase the complexity of the software (and therefore any complexity metric).

3. Private declarations

Declarations using the work "private" tend to reduce complexity in interconnection metrics since they minimize the interface between components of the software. "Limited private" declarations further restrict the interface. The presence of such declarations reduces complexity.

4. Mode restriction

Restricting the mode of parameters to be "in", "out", or "in out" in procedures and "in" only in functions reduces side effects and thus reduces complexity.

5. Exception handling.

The factors that cause exceptions are present in every substantial software project. Exception handling facilities in Ada provide a clean way of treating exceptions. The effect on software metrics is to increase the lines of code, operators and operands and thus increase complexity. However, exception handling is probably the simplest way to write certain segments of code and thus the effect on a metric should be relatively minor.

C. FEATURES WHICH INFLUENCE PORTABILITY.

1. Packages

Packages encourage modularity which is of vital importance in structured design. Clearly a piece of software can be ported to another installation only, if all of the packages called by the software are also ported and there are no name conflicts with existing packages in the new installation.

2. Generics

Generic packages encourage portability by requiring only instantiation to work. However, considerable care must be given to the testing of generic packages, since each instantiation of a package for a new data type should be tested like a new package.

3. Interfaces to other languages

Data abstraction and information hiding are major features of Ada. Clearly any software interface to another language decreases the usefulness of these features. Consider the following example from the Ada Reference Manual [10, p. 217]:

```
package FORT-LIB is
  function Sqrt (X:FLOAT) return FLOAT;
  function Exp (X:FLOAT) return FLOAT;
private
  pragma INTERFACE (FORTRAN, Sqrt);
  pragma INTERFACE (FORTRAN, Exp);
end FORT-LIB;
```

This use of the pragma INTERFACE decreases portability because it uses languages that may not be available in all installations and because the capability for interfaces to other languages need not be made available in all implementations of Ada [10, p. 217]. Clearly, this increases complexity.

4. Machine code insertions

Clearly machine code in an Ada program eliminates portability of that section of the program that includes the machine code. Some machine code can be reorganized by the presence of the pragma INLINE and the use of the predefined library package MACHINE_CODE.

Another source of machine dependent code is the use of specific locations. Examples of these are the for-use, at-mod and use-at constructions. Examples are:

```
for RIGHT_MASK use 2#001#
  (using a bit pattern to mask input

  for example)
FOR_FAILURE_SIGNAL use at 8#4041#
  (using an octal representation
   of a port)
```

and

```
for PIXEL_STORAGE use
  record at mod 4;
    X at SOME_X_VALUE;
    Y at SOME_Y_VALUE;
    COLOR at SOME_COLOR;
    INTENSITY at SOME_LEVEL;
  end record;
```

This last example might be used in a graphics program in which we intend to move a block of pixels and wish to speed up their movement by aligning with byte or word boundaries.

D. FEATURES WHICH INFLUENCE VERIFIABILITY

1. Named parameter association

Consider a generic package

```
generic
X,Y: FLOAT:=0.0;
package POINT is ...
package FIRST_POINT is new POINT
(3.7,2.8);
package SECOND_POINT is new POINT
(X => 3.7,
Y => 2.6);
```

In the package SECOND_POINT, the named parameter association tells us about the names of the parameters as well as their values. The other package describes the values by using positional notation. Named parameter association tends to decrease complexity when used both in this context and in the context of fields of a record.

2. Global variables.

Use of such variables often increases complexity because the availability of a shared variable to two tasks means that neither can assume anything about the order in which the operations of the various tasks are performed except at synchronization points. The syntax for shared variables is

```
pragma SHARED (var_name);
```

Shared variables increase complexity since they increase opportunity for errors.

4. TIME VARYING METRICS

In the previous sections we discussed some standard metrics and the results of their application to a large set of Ada programs. The metrics used all assign a complexity measure to programs and to their component modules, with the primary purpose being the early identification of those programs or modules which are most likely to require changes in the development stage. We now consider the behavior of metrics when they are applied to programs during the entire life cycle rather than restricting attention to the development phase. Note that any metric for Ada programs must take into account the factors mentioned in section 3.

Consider the standard model of the software life cycle. The "maintenance phase" is often caused by either porting code to another machine or by changing specifications to require faster execution, more efficient use of

storage, addition or deletion of processors in a distributed system, demand for a more "friendly" user interface, etc. Changing performance requirement specifications are even more prevalent in a model in which prototypes are developed rapidly with successive modification of the prototypes leading to deliverable products. The evolution of Ada software projects is greatly influenced by factors peculiar to the Ada milieu. These factors are: standardization of the language before the advent of usable compilers, rapidly evolving compiler performance (although compilation speed and quality of code is still not at a very high level compared to more mature languages), and lack of experienced Ada programmers (because of the newness of the language).

It is clear that metrics used only during the coding phase are only an approximation to any quantitative evaluation of the software that will eventually be produced. A metric applied only at one point in the life cycle can only suggest portions of the code that are especially complex. Such metrics are, by their nature, incapable of measuring those portions of the code which are inherently complex. Note also that the goals of such metrics is to aid in the development of code of minimal complexity, regardless of the changing requirements during the useful life of the software project.

The nature of the time variation of metrics is the theme of this section. We intend to return to the topic of evaluating and fine tuning such metrics in a future paper.

With these suggestions in mind we will use the following terminology.

MS = the metric used at the specification stage

MD = the metric used at the design stage

MC = the metric used to evaluate completed code (In the case of a developing system with many prototypes, the metric may be applied to each prototype).

MM = the metric used during the maintenance cycle.

It is natural to ask if the same metric can be used at each one of these four stages in the development of Ada programs; the answer is a resounding no! Among other things, Ada is not a formal specification language since specifications cannot be executed. (Historical note - executable specifications were considered in

several of the interim reports on Ada). Since most software metrics assume that language level is a major factor in the metric, there cannot be a suitable measure MS unless the specification language is fixed. For simplicity, we assume that the specification language is English and that the metric MS is simply defined by

$MS = \text{number of tasks.}$

We are assuming that the number of tasks is known and fixed at this stage. Any dynamically allocated tasks are assumed to be created because of performance or coding criteria in later stages of software development.

Thus the measure at this point is basically an "interconnection metric" since it is based on the number of separately executing components of the program.

The next metric MD is more interesting, because many people consider Ada to be a reasonable design language.

We assume that MD is a metric which should accurately reflect the type of code which will be written during the coding phase. Thus MD and MC should have high correlation. In this paper we assume that $MD = MC$.

The metric MC used at the end of the coding stage (or at the end of the coding stage for each prototype) reflects the standard use of metrics. That is, MC is a static measure of the code. Its use here is to analyze why certain segments of the code were hard to write and to predict and avoid problems which will occur during the maintenance stage. Our empirical evidence (see the graph in Figure 1) supports the metric MC defined by

$MC = 2 * \text{NUMBER_OF_TASKS} + 3 * \text{NUMBER_OF_EXCEPTIONS} + 4 * (\text{NUMBER_OF_ENTRY} + \text{NUMBER_OF_ACCEPT} + \text{NUMBER_OF_SELECT}) + \text{NUMBER_OF_SHARED_VARIABLES} + \text{NUMBER_OF_PRIVATE_TYPES} - (\text{NUMBER_OF_MODES_IN} + \text{NUMBER_OF_MODES_IN_OUT} + \text{NUMBER_OF_MODES_OUT}) / 2.$

The final metric MM is easier to understand. Since each metric has the dual goals of measuring the current product and predicting problem in the future, MM is concerned only with problems in fixing errors, improving performance, adding functionality, and in the code being ported to other host systems. The last three of these reflect changes in the design requirements of the software. At this time we suggest the equation

$MM = MC + \text{NUMBER_OF_PACKAGES} + 4 * \text{NUMBER_OF_TASKS} + 5 * (\text{NUMBER_OF_FOR_USE}) + 3 * (\text{NUMBER_OF_BINARY} + \text{NUMBER_OF_OCTAL} + \text{NUMBER_OF_HEXIDECIMAL}) + \text{NUMBER_OF_AT_MOD}$
for the metric MM.

Consider for example the problem of porting software which allows a maximum of 5 independent tasks from a system with 1 processor to a system with 10 processors. It is likely that such a wealth of resources will cause a major change in the software if performance improvements are not as expected.

5. LIMITATIONS OF ADA SOFTWARE METRICS

The following seems to be the minimum requirement used for any software metric.

Definition. A software metric m for a language L is a function from the set of programs or modules written in the language L to the non-negative real numbers.

Note that this definition says nothing about the input to a metric being a correct program or module. Note also that metrics often have additional properties. For example, Halstead's metric [4] has the property of additivity; that is, if A and B are disjoint modules, then

$$m(A+B) = m(A) + m(B),$$

and this is independent of how A and B are interconnected. Here " $A+B$ " means the joining of A and B in a single program.

The cyclomatic number $e-n+p$ defined by McCabe [8] requires a slightly different analysis. If we assume that the start node of B is identified with a node of A and that no other nodes or edges are common, then the number of nodes of $A+B$ is one less than the sum of the number of nodes of A and the number of nodes of B . Using an obvious notation,

$$n(A+B) = n(A) + n(B) - 1.$$

Also, the number of connected components of A , B , and $A+B$ are related by

$$p(A+B) = p(A) + p(B) - 1$$

and hence we have

$$m(A+B) = m(A) + m(B).$$

It is easy to see that this is true even if A and B have other nodes or edges in common or if A and B have no nodes in common. In any event, McCabe's cyclomatic number satisfies the additivity condition

$$m(A+B) = m(A) + m(B).$$

What about interconnection metrics? Such metrics assume that the primary factor influencing complexity is the interconnections between modules. The number of interconnections increases exponentially as the number of modules increases. Hence the additivity condition is replaced by the condition

$$m(A+B) \geq m(A) + m(B)$$

where "A+B" now represents a program with modules A and B (which of course may also be composed of other modules). Thus these metrics fail to have the property of "subadditivity", much less additivity. This is a major reason for the lack of a well-defined theory of such metrics for general use in evaluating software written in most programming languages.

The situation for Ada software metrics is somewhat different from the situation for general programming language metrics. As was indicated in section 3, the interfaces between component modules are tightly controlled by the standard interfaces, such as restricting modes in functions to "in", "out", or "in out". For separate packages, there are no common variables unless the common variables are declared by the pragma SHARE in each package. Thus Ada metrics when applied to structured code have the additivity property that the measure of a program composed of two separate program units is equal to the sum of the measures of the two program units. Hence it is reasonable to suspect that there is some mathematical structure underlying the theory of Ada software metrics of the type presented here.

It is important to distinguish the factors that are to be measured by a metric. Such factors include the language level, underlying complexity of the problem, experience and ability of the programmer, relative frequency of expected errors in certain code segments, and degree of difficulty in implementing the changes in the next phase of the software life cycle. Metrics are applied to code for the purpose of evaluating the code and predicting those portions which will be troublesome in the next phase of the life cycle.

With these points in mind, we make the following definition. We define an Ada software metric scheme to be a quadruple (ms, md, mc, mm) of functions called metrics whose range is a subset of the set of real numbers and which satisfy the following conditions:

a. The domain of ms is the set of all possible specification language (which may be English, formal specification language with executable code, or something in between).

b. The domains of md, mc, and mm are the set of all possible Ada programs (assuming Ada is the design language).

c. Each of the metrics md, mc, and mm is the sum of two non-negative functions. These functions are the P-measures which represent the metric applied only to those portions of the code that are direct translations of Pascal programs and the A-functions which represent the complexity of the code caused by Ada-specific features not present in the language Pascal.

Defining a metric scheme as a collection of four metrics seems somewhat redundant at first glance. Much of the research in software metrics involves a search for a single measurement to be used at all times. However, Dunsmore and Gannon [3] performed an interesting experiment in the use of global variables and formal parameters in communicating between various modules. They observed that global variables tend to decrease errors during program development but that formal parameters tend to decrease errors during the maintenance phase. Their results support the need for different metrics during different phases of the life cycle.

Recall that in section 3 we observed that any metric for Ada programs must consider the typing, tasking, packages, modes, generics, exceptions and other special features of Ada. Clearly these must be present in the A-function since they are not available in the language Pascal. We are now ready to state and prove the major result which shows the limitation of Ada software metrics.

THEOREM It is impossible for any metric m applied to any stage of the software life cycle of Ada programs to be able to predict the complexity of the code. In fact, given any Ada software metric scheme, there is a program E for which each of the metrics in the scheme has a fixed value when applied to the code but the measures applied to E during execution can grow arbitrarily large.

Proof. Consider a program E which involves code of the form
task type T_TYPE is

.....
type TASK_POINTER is access T_TYPE;

Any metric applied to the program E at any stage in the software life cycle will assign a fixed non-negative number to E. However, tasks can be spawned dynamically at run-time. The number of tasks, and hence the number of interconnections between tasks, can be made arbitrarily large at run time.

Hence any complexity measure, when applied to the code-data pair during runtime, can be made arbitrarily large even though the value of the measure on E before run time was fixed.

This theorem shows that it is impossible for any metrics which can be applied to every program at any stage of the life cycle to be able to precisely predict problems in complexity of the code. There are several possible options to partially resolve this situation.

1. Continue to apply these metrics (or later, more polished versions) to Ada programs recognizing that these metrics cannot give complete information on all Ada programs.

2. Apply these metrics only to a subset of all possible Ada programs. This procedure is analogous to the situation in the branch of mathematics called measure theory where measures are not applied to all sets.

3. Partition the metrics ms, md, mc and mm into a collection of functions, not all of which can be applied to all programs. These functions will incorporate the ideas in section 3.

6. CONCLUSIONS AND DISCUSSION OF FURTHER RESEARCH

This paper had two main themes: development of software metrics for Ada programs and determining the limitations of such metrics. Research on the development of metrics involved examination of several classical metrics on a sample of short Ada programs. Results obtained suggested some metrics for Ada programs that are incorporated into Ada metric schemes. These metric schemes are quadruples of metrics (md, md, mc, mm) which are applied during the specification, design, coding, and maintenance phases of the life cycle. Research will

continue into such metric schemes and their validity as predictors of problems with programs at various points in the software life cycle.

The second main theme of this paper was the theoretical limitations of such metrics. We proved a theorem indicating that no metric applied to static code can predict code complexity for programs which change their complexity at runtime. The proof of the theorem is based on a particular Ada concept. Future research in this area will concentrate on extending this theorem to determine which other properties of Ada programs cause similar difficulties with Ada metrics.

Representing metrics as the sum of P-metrics (for Pascal-like program features) and A-metrics (for program features available in Ada but not in Pascal-like languages) is a first step in this research.

Acknowledgement

This research was partially supported by the Army Research Office under grant number DAAL-03-86-G-0085.

REFERENCES

1. Basili, V. and Wu, L., "Structure Coverage Tools for Ada Software Systems", to appear in Proceedings of the 1986 Joint Conference of the National Conference on Ada Technology and Washington Ada Symposium.
2. Carrio, M., "The Technology Life Cycle and Ada", Proceedings 4th Annual National Conference on Ada Technology, March 19-20, 1986, Atlanta, GA., 75-82.
3. Dunsmore, H.E. and Gannon, J.D., "Analysis of the Effect of Programming Factors on Programming Effort", J. Syst. Software 1,2 (Feb. 1980), 141-153.
4. Halstead, M.H., "Elements of Software Science", Elsevier North-Holland, New York, 1977.
5. Kafura, P. and S. Henry, "Software Quality Metrics Based on Interconnectivity", J. Systems and Software, 2 (1981), 121-131.

6. Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., Adler, M.A. and M.A. Gray, "Problems with Software Complexity Measurement", Proc 1985 ACM Computer Science Conference, March 1985, Cincinnati, Ohio, 340-347.

7. Keller, S.E. and J.A. Perkins, "An Ada Measurement and Analysis Tool", Proc. 3rd Annual National Conference on Ada Technology, March 1985, Houston, Texas, 188-196.

8. McCabe, T.J., "A Complexity Measure", IEEE Trans. Software Engineering, SE-2, 1976, 308-320.

9. Perkins, J.A., Lease, D.M. and S.E. Keller, "Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada", Proc 4th Annual National Conference on Ada Technology, March 1986, Atlanta, GA, 67-74.

10. "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A, U.S. Dept. of Defense, 1983.

11. Reynolds, R.G. and D. Roberts, "PARTIAL: A Tool to Support the Metrics Driven Design of Ada Programs", Proc 15th ACM Comp Science Conference, Feb. 1986, 213-219.

12. Ramamoorthy, C.V., "Languages for Software Engineering", keynote address at IEEE 1986 International Conference on Computer Languages, Miami, Fla., October 28-30, 1986.

13. Taylor, R.N. and T.A. Standish, "Steps to an Advanced Ada Programming Environment", IEEE Trans. Softw. Eng. SE-11, 3(1985), 302-310.



BIOGRAPHICAL SKETCH

Ronald J. Leach is a Professor of Systems and Computer Science at Howard University. He has BS, MS, and PhD. degrees from the University of Maryland in Mathematics and a MS degree from Johns Hopkins University in Computer Science. His current research interests include software engineering, analysis of algorithms, computer graphics and user interfaces.

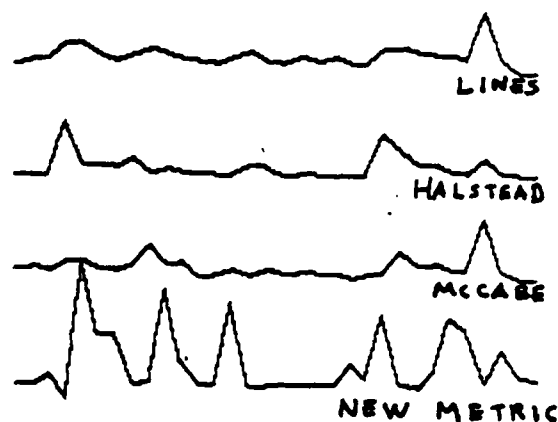


FIGURE 1

FORMAL CONCURRENT TASKING PARADIGMS IN THE DESIGN OF ADA PROGRAMS

Ronald J. Leach
Darlene Bond

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

ABSTRACT

A major feature in the design of Ada was the high level support of concurrent tasks. Concurrent tasking is an essential feature of embedded systems in most environments. In this paper we examine the state of Ada education in the support of concurrent tasking. The tasking examples are compared to formal models in C.A.R. Hoare's CSP (Communicating Sequential Processes) system. The resulting information is compared with actual use of tasking programs in the Ada literature and in industry and government. Particular attention is paid to the treatment of non-determinism in tasking programs and in formal models.

INTRODUCTION

In order to maximize the effectiveness and efficiency of a program, a programmer must begin the program development with a good program design structure. Programs are made up of several types of building blocks. Programs without concurrent execution of tasks use the standard sequential building blocks of procedures, functions, and modules. Programs involving concurrent execution use these building blocks and the additional block of a task which is usually a collection of the sequential building blocks. The use of concurrent tasking in programs greatly increases the potential for error in programs and thus causes great difficulty during all phases of the software life cycle. Errors which occur at many phases of the software life cycle and costs which increase exponentially are major features of the "software crisis". It is clear that the current "software crisis" will get even worse since most of the existing problems have been with systems which do not involve much concurrent execution.

Abstraction and information hiding are major techniques of software engineering that are used to address some of the problems with software. Indeed, the relative ease in which information hiding and abstraction of data are implemented in the Ada language is a major reason for the success of Ada. It is clear from the success of these language features that there is a need for formalism in the area of concurrent programming in Ada.

Perhaps the most common paradigm for design of programs involving concurrent tasks is C. A. R. Hoare's Communicating Sequential Processes (CSP). It is reasonable to ask if Hoare's abstract models of CSP involving concurrency are applicable and effective tools in program design. Ada was originally intended for use with embedded systems and concurrent tasking and to incorporate principles of good software engineering; it is appropriate at this point to examine how these two ideas work together in practice. This research was conducted to see if the current state of use of abstract models of Ada programs involving concurrent tasking is sufficiently well-understood to be used in providing a basis for Ada program design. Thus this research represents an assessment of how well the use of tasking and formal models is supported in the existing Ada educational community.

The first step in approaching this problem was to collect data. The data was initially collected from the published literature of Ada programs including textbooks, lecture notes, and conference proceedings. We chose 17 texts from the library; the selection criterion was actually having the book on the shelf and not in circulation at the time that the data was gathered. We feel that this is a representative sample of the use of tasking in the existing Ada textbook literature. Programs which involved tasking were extracted and examined to see

ered. We feel that this is a representative sample of the use of tasking in the existing Ada textbook literature. Programs which involved tasking were extracted and examined to see which, if any, of Hoare's models could be applied to the programmers' method of executing the task or tasks. The results were tabulated to see which models were applied in these programs.

The textbooks examined fall into two categories: limited amounts of tasking (including none at all) and considerable emphasis. A total of 819 programs from all textbooks were examined, with only 114 or 13.9% having any concurrent tasks. We note that most of the programs involving tasking (36) were found in a single reference [7]. Of the programming samples obtained from textbooks in the first category, there were only 32 programs involving tasking out of a total of 730 or 4.3%. The PIQ model of concurrent execution of tasks with no communication between the tasks was found most often, with a total of 14 instances, of which 12 involved only two tasks. The repetition of tasks, which is denoted abstractly as *P, was the next most frequently found, with seven instances. The next most frequent model occurring is the P;Q model in which the tasks actually are executed in order, a total of 6 instances. Hoare distinguishes 29 distinct models for tasking involving two tasks; only 8 of them or 27.5% are represented in the texts. In table 1 below, we summarize our search of the textbook literature, some examples of student programs, and sample programs that are available in the non-textbook Ada literature. Note that we show the number of tasks in each example and therefore do not quite agree with all of Hoare's categories, since Hoare only lists the possibilities for the execution of two concurrent tasks in his explicit listing of possibilities.

The textbooks [5] and [7] had much more emphasis on concurrent programming as the titles "Concurrent Programming in Ada" and "Parallel Programming in ANSI Standard Ada" would indicate. There were a total of 89 programs presented with tasking evident in 42 or 51.7%. Here the range of programs is much wider including examples of (PIQ)*, \$(P sub 1 IIP sub 2 ..IIP sub n)*, \$\$, PIQ with Q of the form RIIS;T and several other models.

The next set of data was obtained from student programs. The intention here was to measure the level in which tasking is used in

such programs. A preliminary experiment involving the examination of 12 student programs using concurrency indicated that the sequential execution of tasks predominated, with 8 uses of the P;Q model of sequential non-communicating tasks, 2 with parallel execution of non-communicating tasks (PIQ), one with the (*(P;Q))) model of repeated sequential tasks, and one with the P;*Q model of task followed by repetition of a sequential task. Some of the programs obtained from students followed the P;Q model which describes the execution of two processes or tasks which are executed sequentially. Some observations about the difficulties encountered by students in the development and execution of these programs was made in [14].

An additional data set was obtained from the existing published non-textbook literature. This data was obtained from the newsletter AdaLetters (including its predecessor), proceedings of several Ada conferences, the Journal of Pascal, Ada, and Modula-2, materials from a variety of Ada short courses, and the Ada Repository. Again in this case, few of the sample programs supported the more complex models.

The most common example of Ada tasking programs was the consumer-producer problem which was presented in various forms. Many texts, especially [7], gave several different solutions to this problem. In some instances, there were two relatively different coding solutions to the same abstract model, even though the two models appeared to have the same CSP representation. We intend to pursue this subject in future work.

The remaining data was collected from a small set of programs actually used in industry and government. Some of these programs make elaborate and extensive use of tasking while of course others do not. The data collected is incomplete at this point because of the difficulty in obtaining samples of actual proprietary code. We do not expect that this data will ever be complete or that it will represent the precise percentages of use of Ada tasking in Ada programs. Instead, we consider it as an example of how Ada tasking paradigms are used in a few hopefully representative Ada applications.

TABLE 1: READILY AVAILABLE INFORMATION ON TASKING. NOTE THAT SOME OF THE DATA COULD ALSO BE CONSIDERED AS MORE COMPLICATED CSP MODELS

CSP MODEL	NUMBER OF OCCURRENCES			
	BOOKS	BOOKS WITH TASKING	STUDENT PROGRAMS	LITERATURE
P Q	12	18	2	8
P[] Q	1			
*P	7			
P;Q	2	2	7	
b*P		1		
P//Q	1			3
x:A->P(x)	1			
A->P	1			2
*P;Q	1			
P Q R	2	8		3
P;Q;R	1			
P//Q R	1	1		
P[]Q[]R	1			
P Q R S	1	2		4
P;*Q	1			
((P;Q)*)*			1	
(P Q R)*		1		
(P Q)*		3		
P1 .. Pn		2		
(P1 .. Pn)*		1		
((Q/P1) .. (Q/Pn))*		2		
TIMED TASKS				2

SUMMARY AND CONCLUSION

It is clear that the quality of information available to beginning and intermediate Ada programmers and designers about tasking is quite limited and does not address the full range of potential tasking uses. The actual problem is much worse than this because Hoare's CSP models do not allow for time constraints such as delays and fixed waits. Such factors are critically important in situations such as the FAA control system or indeed in any system that must perform in real time.

It is well-known that even experienced programmers have considerable difficulty in writing programs which involve any degree of concurrency. We recommend the following solutions.

1. At the preliminary level of education; that is, in the undergraduate and graduate programs of colleges and universities, the amount of instruction in concurrent pro-

gramming must be increased. This instruction should be done over a variety of courses so that students see these ideas in a number of contexts.

2. Textbooks in the language Ada must include a wider variety of tasking programs including more of Hoare's CSP models. While the amount of tasking information need not be as much as in [7], it must be increased in order to make sophisticated knowledge of Ada tasking available to as many students as possible.

3. Continuing education for the professional should include a comprehensive study of tasking in Ada. This is not appropriate for the first introduction, which should be limited to the fundamental features of the language and Ada software engineering with only a brief introduction to tasking. Second courses should give views of many abstract models of tasking by means of many

different examples. We note that this is being done at Pennsylvania State University (Capitol Campus) and at Computer Science Corporation (Moorestown).

4. In the absence of high quality educational opportunities or having existing personnel already well trained in Ada tasking, management must choose between using special expertise from outside the organization and restricting the tasking to the simple models supported by most of the existing texts.

Acknowledgement

Research of both of the authors of this paper was partially supported by the U.S. Army Research Office under grant number DAAL-03-86-G-0085.

REFERENCES

1. Amoroso, S. and G. Ingargiola, *Ada : An Introduction to Program Design and Coding*, Pittman, Boston, 1985.
2. Ausnit, C. et al, *Ada in Practice*, Springer-Verlag, New York, 1985.
3. Booch, G., *Software Engineering with Ada*, Benjamin Cummings, Menlo Park, California, 1983.
4. Buhr, R.J.A., *System Design with Ada*, Prentice-Hall, Englewood Cliffs, 1984.
5. Burns, A. *Concurrent Programming in Ada*, Cambridge University Press, Cambridge, England, 1985.
6. Caverly, P. and P. Goldstein, *Introduction to Ada: a Top-down Approach for Programmers*, Brooks/Cole, Monterey, California, 1986.
7. Cherry, G., *Parallel Programming in ANSI Standard Ada*, Reston Publishing Co., Reston, Va, 1984.
8. Cohen, N., *Ada as a Second Language*, McGraw-Hill, New York, 1986.
9. Downes, V.A. and S.J. Goldsack, *Programming Embedded Systems with Ada*, Prentice-Hall, London, 1982.
10. Gehani, N., *Unix Ada Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
11. Haberman, A.N. and D.E. Perry, *Ada for Experienced Programmers*, Addison-Wesley, Reading, Massachusetts, 1983.
12. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
13. Katzan, H.Jr., *Invitation to Ada*, Petrocelli Books, New York, 1984.
14. Leach, R., *Experiences Teaching Concurrency in Ada*, AdaLetters, 1987.
15. Mohnkern, G.L. and B. Mohnkern, *Applied Ada*, Tab Professional and Reference Books, Blue Ridge Summit, Pennsylvania, 1986.
16. Pyle, I.C., *The Ada Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
17. Texel, P.P., *Introductory Ada: Packages for Programming*, Wadsworth, Belmont, California, 1986.
18. Wegner, P., *Programming with Ada: An Introduction by Means of Graduated Examples*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
19. AdaLetters, various issues 1983-1987.
20. Journal of Pascal, Ada, and Modula-2, various issues 1986-1987.
21. Proceedings of the First Annual National Conference on Ada Technology
22. Proceedings of the Second Annual National Conference on Ada Technology
23. Proceedings of the Third Annual National Conference on Ada Technology
24. Proceedings of the Fourth Annual National Conference on Ada Technology, Atlanta, GA, March 19-20, 1986.
25. Proceedings of the Joint Ada Conference, Washington, DC, March 16-19, 1987.

Darlene Bond received a Bachelor of Science degree in Psychology from Howard University in 1983. She is a graduate research assistant in the School of Engineering's Systems and Computer Science Department at Howard and will receive a M.S. in Computer Science in May, 1988. Her professional interests include systems programming, computer graphics, software engineering, and artificial intelligence.

Ronald J. Leach is a Professor in the Department of Systems and Computer Science at Howard University. His research interests include software engineering, computer graphics and concurrent computing.

The Actual Complexity of Parallel Evaluation of low Degree

Polynomials

Ronald J. Leach

O. Michael Atogi

Razeyah R. Stephen

Department of Systems & Computer Science

School of Engineering

Howard University

Washington, D.C. 20059

ABSTRACT

We consider several sequential and parallel algorithms for the evaluation of polynomials of low degree, with particular emphasis on those that are used frequently in computer graphics. A complete accounting of computation times for the speed-up and efficiency of these algorithms is reported. The results are compared to standard estimates of these quantities for single and multi-processors using classical complexity theory. A simulator which is configurable to several parallel architectures is used to provide validation of the results obtained.

1. INTRODUCTION

It has been clear for several years that major improvements in execution time for many programs will require extensive use of parallel processing. Many papers have been written exploring the computational complexity of algorithms which are developed for parallel computation. The complexity is usually measured on some abstract machine which has certain properties that are assumed to be somewhat realistic. In general, these theoretical results do not have a particularly good correlation with observed execution times on actual hardware realizations of these abstract parallel machines. The paper [4] is a typical example. Typically the quality of a parallel algorithm is measured by two quantities called the efficiency and the speed-up. Speed-up is defined by the formula

$$S(p) = T(1)/T(p)$$

while efficiency is defined by

$$E(p) = S(p)/p.$$

where p denotes the number of processors. These measures are well-defined for any given algorithm provided that all of the times involved in the computation are taken into account. Typically, the "best" sequential algorithm is used to compute $T(1)$. The articles [2], [5], and [9] present various views of what the actual speed-up of an algorithm is. In [9], Parkinson claimed that a particular parallel algorithm for adding two vectors has efficiency greater than 1. This analysis was disputed by [2] and [5], where the authors indicated that certain implicit assumptions were made by Parkinson. They described other factors involving actual performance on any hardware realization of an abstractly described parallel computer.

The controversy over this simple algorithm suggests that some of the classical results of arithmetic complexity theory be reviewed from the point of view of the actual times needed for performance of needed operations in an arithmetic computation. In this paper we are concerned with the evaluation of polynomials. We will consider a number of algorithms for evaluation of low degree polynomials and obtain estimates of run time speed using techniques of classical arithmetic complexity theory. The actual numbers of memory accesses, register moves, index changes, arithmetic operations, and inter-process communications will be given and will be translated to actual efficiency and speed-up for a number of parallel architectures.

We restrict our attention to low degree polynomials in this paper for several reasons. First, the actual time costs of all of the operations are apparent in low degree polynomial evaluation. Second, our intention was to compare actual to theoretical results. It is easy to do this for low degree polynomials. Finally, real-time graphics, which is one of the most computationally intensive fields and is a typical target for parallel computation, is concerned almost exclusively with the evaluation of cubic polynomials or quotients of cubic polynomials when solid objects are displayed. For additional information, see the reference [3] from which the following discussion of the use of low degree polynomials in computer graphics is taken.

Low degree polynomials appear naturally in computer graphics in the following context. Suppose that the graph of some surface is to be displayed. The surface is approximated by a collection of *patches* which are defined by a collection of quadruples of points. These quadruples may represent points on the surfaces or some combination of points on the surface or points of tangency, or points that are chosen to represent the smoothness of the surface without lying precisely on the surface. The patch is given in parametric form $x = x(u,v)$, $y = y(u,v)$, $z = z(u,v)$, where u and v are restricted to lie in some region which is almost always the unit square $0 \leq u, v \leq 1$. The functions $x(u,v)$, $y(u,v)$, and $z(u,v)$ are either products of polynomials of degree at most 3 or are quotients of such products. The reason for this is technical and involves having exactly the minimum number of degrees of freedom to allow for smoothness when two patches are joined.

The graph of the surface is approximated near the patch by the following algorithm.

```
FOR each patch DO
  /* evaluate the patch for curves of constant u */
  for (u = 0; u ≤ 1; u = u + increment)
    move_abs_3(x(u,0), y(u,0), z(u,0));
    for (v = increment, v ≤ 1, v = v + increment)
      line_abs_3(x(u,v), y(u,v), z(u,v));

  /* evaluate the patch for curves of constant v */
  for (v = 0; v ≤ 1; v = v + increment)
    move_abs_3(x(0,v), y(0,v), z(0,v));
    for (u = increment, u ≤ 1, u = u + increment)
      line_abs_3(x(u,v), y(u,v), z(u,v));
```

Each patch requires $2(\text{increment})^2$ evaluations of the functions $x(u,v)$, $y(u,v)$, $z(u,v)$ for a total of $6(\text{increment})^2$ evaluations of functions. A typical realistic picture in computer graphics may contain between 500 and 2000 or more patches and may require values of *increment* of .001. This corresponds to 1.2×10^{12} possible function evaluations of the three component functions x , y , and z . Clearly a critical factor is the speed in which the functions can be evaluated. More importantly, the large number of computations needed to solve this problem strongly suggests a need for some of the computations to be performed in parallel.

2. SOME STANDARD RESULTS FROM COMPLEXITY THEORY

We recall a few results from classical arithmetic complexity. Let n be a positive integer and consider the polynomial

$$p(x) = a_0 + a_1x + \dots + a_nx^n.$$

Suppose that we wish to evaluate this polynomial on a single processor. To evaluate $p(x)$ for a given x requires the computation of x^n and all lower powers of x . The most obvious algorithm requires $2n - 1$ multiplications and n additions. A faster procedure is Horner's method where we write

$$p(x) = (\dots((a_nx + a_{n-1})x + a_{n-2})x \dots) + a_0.$$

This requires n multiplications and n additions. Knuth [6] gives an algorithm due to Belaga which requires $\left\lfloor \frac{n}{2} \right\rfloor + 2$ multiplications and n additions. See [7] and [10] for other results in this area.

For parallel evaluation of polynomials, some major results are due to [7], [8], and [10]. These results suffer from some of the same difficulties as the previously mentioned work on the efficiency and speed-up. See the paper [1] for an analysis of some of the difficulties involved.

3. PARALLEL ALGORITHMS USED FOR POLYNOMIALS IN A SINGLE VARIABLE

In this section, we describe algorithms (both sequential and parallel) for evaluation of low degree polynomials in one variable. In order to conserve space, we have omitted each case of Horner's algorithm for single processors and have followed the notational conventions of using the terminology of the original papers with "pre-processing" and not explicitly writing obvious communications between processors.

A2.2: (Horner's method, quadratic polynomial, two processors)

PROCESSOR 1	PROCESSOR 2
a_2x	a_1x
a_2x^2	$a_1x + a_0$
$a_2x^2 + a_1x + a_0$	

for second degree polynomials,

A3.1a: (Knuth's method, cubic polynomial, single processor)

$$y = x + c$$

$$w = y^2$$

$$y_2 = w - \alpha_1$$

$$z = a_3 y$$

$$z = z + \beta_0$$

$$zy_2$$

A3.2: (Horner's method, cubic polynomial, two processors)

PROCESSOR 1	PROCESSOR 2
x^2	x^2
$a_2 x^2$	$a_3 x^2$
$a_2 x^2 + a_0$	$a_3 x^2 + a_1$
	$(a_3 x^2 + a_1)x$
$a_2 x^2 + a_0 + (a_3 x^2 + a_1)x$	

A3.2a: (Knuth's method, cubic polynomial, two processors)

PROCESSOR 1	PROCESSOR 2
$y = x + c$	$y = x + c$
y^2	$a_3 y$
$y^2 - \alpha_1$	$z = a_3 y + \beta_0$
$z(y^2 - \alpha_1)$	

for third degree polynomials and

A4.1a: (Knuth's method, quartic polynomial, single processor)

$$y = x + c$$

$$w = y^2$$

$$w - \alpha_1$$

$$a_4 y$$

$$a_4 y + \alpha_0$$

$$(a_4 y + \alpha_0)y$$

$$z = (a_4 y + \alpha_0)y + \beta_0$$

$$z(w - \alpha_1)$$

A4.2: (Horner's method, quartic polynomial, two processors)

PROCESSOR 1	PROCESSOR 2
x^2	x^2
a_4x^2	a_3x^2
$a_4x^2 + a_2$	$a_3x^2 + a_1$
$(a_4x^2 + a_2)x^2$	$(a_3x^2 + a_1)x$
$(a_4x^2 + a_2)x^2 + a_0$	
$(a_4x^2 + a_2)x^2 + a_0 + (a_3x^2 + a_1)x$	

A4.2a: (Knuth's method, quartic polynomial, two processors)

PROCESSOR 1	PROCESSOR 2
$x + \alpha_0$	$x + \alpha_2$
$z = (x + \alpha_0)x$	delay
$y = z + \alpha_1$	delay
	$x + \alpha_2 + y$
	$(x + \alpha_2 + y)y$
	$(x + \alpha_2 + y)y + \alpha_3$
	$((x + y + \alpha_2)y + \alpha_3)\alpha_4$

and

A4.2b: (Knuth's method with modifications, quartic polynomial, two processors)

PROCESSOR 1	PROCESSOR 2
$y = x + c$	$y = x + c$
y^2	a_4y
$y^2 - \alpha_1$	$a_4y + \alpha_0$
delay	$(a_4y + \alpha_0)y$
	$z = (a_4y + \alpha_0)y + \beta_0$
	$z(y^2 - \alpha_1)$

for fourth degree polynomials.

These algorithms were implemented on a variety of distributed systems with processing elements of several different architectures. The results are given in the next section.

In each case, the processors used in the distributed system were identical. The experiment was repeated for each algorithm so that some independence of processors was obtained. Assembly code was not optimized by any unusual tricks but was given with the intention of imitating typical code generated by a compiler. Thus the results should be typical of the situation actually encountered in practice, especially since the general problem of efficiently using registers for computation on an arbitrary computer

is extremely difficult.

Suppose we define a processing element as a triple (CPU, memory, single data channel-in, single data channel-out). The results of this paper are described in the following theorem. We will need to use the following notation:

t_{add} = time for addition/subtraction,
 t_{mult} = time for multiplication/division,
 t_{index} = time for array index access,
 t_{mem} = time for memory access,
 t_{loop} = time for loop control
 t_{comm} = time for communication between two adjacent processors.

Theorem 1 : Let F be an arbitrary polynomial of degree 2, 3, or 4 in a single real variable u . Suppose also that F has real coefficients. Let $E(p,n)$ and $S(p,n)$ be the efficiency and speed-up for the problem of evaluating $F(u)$ at a real variable u on a p -processor non-pipelined system for a polynomial of degree n . Then

$$\begin{aligned}
 S(2,2) &\leq (2t_{add} + 2t_{mult} + 5t_{mem} + 2t_{index}) / (t_{add} + 2t_{mult} + 3t_{mem} + t_{index} + t_{comm}) \\
 E(2,2) &= S(2,2)/2 \\
 S(2,3) &\leq (3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) / (2t_{add} + 3t_{mult} + 4t_{mem} + 2t_{index} + t_{comm}) \\
 S(2,3) &\leq (3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) / (2t_{add} + 2t_{mult} + 5t_{mem} + 2t_{index} + t_{comm}) \\
 E(2,3) &= S(2,3)/2 \\
 S(2,4) &\leq (4t_{add} + 4t_{mult} + 9t_{mem} + 4t_{index}) / (3t_{add} + 3t_{mult} + 6t_{mem} + 2t_{index} + t_{comm}) \\
 E(2,4) &= S(2,4)/2
 \end{aligned}$$

Proof.

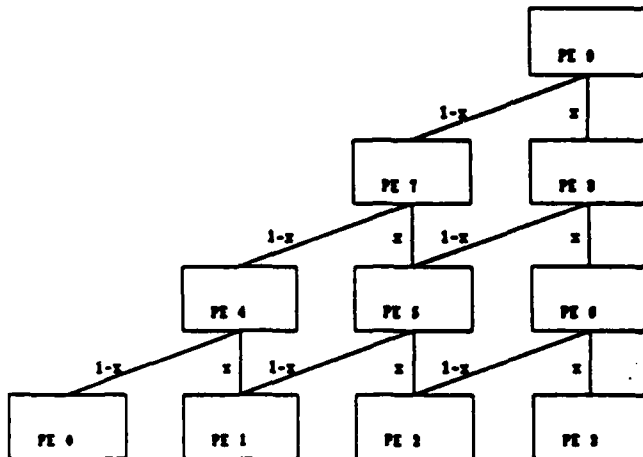
Algorithms A2.1, A3.1, and A4.1 represent Horner's method on one processor. Note also that we can reduce the number of index accesses by one if we note that a_0 is typically in the first memory cell devoted to the array of coefficients. We have the following table which describes the total time needed for evaluation of the polynomial F .

ALGORITHM	TOTAL TIME
A2.1	$2t_{add} + 2t_{mult} + 5t_{mem} + 2t_{index}$
A2.2	$t_{add} + 2t_{mult} + 3t_{mem} + t_{index} + t_{comm}$
A3.1	$3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}$
A3.1a	$3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}$
A3.2	$2t_{add} + 3t_{mult} + 4t_{mem} + 2t_{index} + t_{comm}$
A3.2a	$2t_{add} + 2t_{mult} + 5t_{mem} + 2t_{index} + t_{comm}$
A4.1	$4t_{add} + 4t_{mult} + 9t_{mem} + 4t_{index}$
A4.1a	$4t_{add} + 4t_{mult} + 9t_{mem} + 4t_{index}$
A4.2	$3t_{add} + 3t_{mult} + 6t_{mem} + 2t_{index} + t_{comm}$
A4.2a	$4t_{add} + 3t_{mult} + 9t_{mem} + 4t_{index} + t_{comm}$
A4.2b	$3t_{add} + 3t_{mult} + 8t_{mem} + 2t_{index} + t_{comm}$

Examination of the table completes the proof of the theorem.

Remarks:

1. These bounds on the efficiency and speed-up are not increased if we allow each of the processors to have registers with faster access time than normal memory access.
2. The performance algorithms can be speeded up if we feed in data via a pipeline. This situation was not described by the theorem since it violates the single data channel-in and single data channel-out requirement. In particular, our results do not apply to algorithms such as deCasteljau's method pictured below for computing cubics. Note that the cubics produced are not in standard form.



In deCasteljau's method, the values of P_0 , P_1 , P_2 , and P_3 are given to the processing elements in the bottom row. These values are multiplied by x and sent to the processor pictured above at the same time that they are multiplied by $1 - x$ and sent to the processor at right. This simultaneity is due to the pipelined architecture and the special design of the processing elements. In the next step, this is repeated in the next row of processing elements P_4 , P_5 , P_6 . At the same time, the processors in the bottom row are sent new values of x to continue the process. A similar thing happens when data flows up from the first row to the second row and from the second row to the top row. Let us ignore the time t_{index} for indexing the four values that are given to the processing

elements in the bottom row. The steady state speed-up and efficiency of this algorithm are each much greater than that of the obvious sequential analog because of the tremendous decrease in memory access and pipelining. In fact, for a single value we have a parallel time of $5t_{mem} + 3t_{comm} + 3t_{mult} + 3t_{add}$ as opposed to $42t_{mem} + 12t_{mult} + 12t_{add}$ for the sequential method. Evaluation of N points using this pipeline involves $(N + 2)(t_{mem} + t_{mult} + t_{add})$ as opposed to the sequential time which is $42Nt_{mem} + 12Nt_{mult} + 12Nt_{add}$. In this case, the speed-up is at least $12N/(N+2)$ and the efficiency is at least 1 for N larger than 10. This provides another example of the well-known fact that efficiencies greater than one are easy to obtain if we consider poor sequential algorithms.

3. Note that the trivial evaluation of a polynomial of degree one takes only two arithmetic steps (one multiplication and one addition), 3 memory accesses and one index time and thus this evaluation can be done fastest on a single processor.

4. PARALLEL ALGORITHMS FOR POLYNOMIALS IN TWO VARIABLES

In this section we will consider three types of algorithms: parallelized versions of the algorithms presented in the previous section, parallelized versions of matrix algorithms, and pipelined algorithms based either on the deCasteljau method or on the method of finite differences.

The evaluation of polynomials in two variables can be described in terms of the evaluation of polynomials in a single variable. A simple way to do this is to recognize that a polynomial F in two variables u and v can be considered as a polynomial P in one variable v whose coefficients are polynomials Q_i in the variable u . Using this idea, the time for sequential evaluation of a polynomial F where u and v appear to at most the third power is four times the time for the evaluation of a polynomial in a single variable u (since there are four polynomials as coefficients) plus the time for evaluation of a polynomial in v whose coefficients are the Q_i . This is 5 times the values of the time given in the previous section. Using Horner's method as a starting point, the time for evaluation is

$$5(3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) + 8t_{mem} + 4t_{index}$$

where the additional terms are obtained from accessing the Q_i .

A matrix method is frequently used for sequential evaluation of polynomials in two variables on a single processor. This method involves the quadratic form UMV^T where U and V represent the row vectors $1, u, u^2, u^3$ and $1, v, v^2, v^3$, M represents the matrix of coefficients, and the superscript T indicates the transpose. This involves 4 multiplications for the powers of the variables, 20 multiplications involving coefficients, 15 additions, 55 memory accesses, and 40 index accesses. The total can be reduced by removing redundant multiplications by 1 to

$$15t_{add} + 19t_{mult} + 45t_{mem} + 40t_{index}, \quad (4.1)$$

which is slightly slower than the first sequential example.

The computations in this algorithm can obviously be speeded up by performing some of them in parallel. For example, each of the 4 multiplications of a row and a column vector can be done in parallel with the last results being communicated to a single processor. Using 4 processors, the time is reduced to

$$6t_{add} + 6t_{mult} + 18t_{mem} + 16t_{index} + 4t_{comm}. \quad (4.2)$$

and the speedup is

$$\frac{15t_{add} + 19t_{mult} + 45t_{mem} + 40t_{index}}{6t_{add} + 6t_{mult} + 18t_{mem} + 16t_{index} + 4t_{comm}}.$$

In the remainder of this section, we will consider the extent to which the times in equation (4.2) can be improved if some of the computations are done in parallel.

We will use the term *bicubic polynomial* to denote a polynomial in two variables with real coefficients such that each variable appears to at most the third power.

Theorem 2 : Let $P(u, v)$ be a bicubic polynomial in the real variables u and v . The speedup in the time to evaluate $P(u, v)$ using two processors and no matrix methods is at most

$$\frac{15t_{add} + 15t_{mult} + 43t_{mem} + 19t_{index}}{9t_{add} + 9t_{mult} + 24t_{mem} + 10t_{index} + t_{comm}} \quad (4.3)$$

and the efficiency is at most 19/10. Using four processors, the speedup is at most

$$\frac{15t_{add} + 15t_{mult} + 43t_{mem} + 19t_{index}}{6t_{add} + 6t_{mult} + 15t_{mem} + 10t_{index} + 4t_{comm}} \quad (4.4)$$

and the efficiency is at most 43/15.

Proof.

In the algorithms given for evaluation of a polynomial in a single variable presented in the previous section, all of the coefficients were assumed to have been loaded into the memory of the appropriate processor. In addition, the time for the pre-processing of the original coefficients in Knuth's algorithms (A3.1a, A3.2a, A4.2a, and A4.2b) was ignored since it was presumed to have been small compared to the large amount of time needed for evaluation of the polynomial at many points. Neither of these assumptions may be used directly in the case of polynomials in many variables since the dependence of coefficients of a polynomial pre-processing and loading of coefficients has not generally been done prior to the evaluation of the polynomial.

For two processors, there are two basic organizations possible: use a parallel algorithm for each evaluation or use sequential algorithms for the polynomials in one variable, using each processor in a sequential manner, with communication between processors at the end. Using the parallel Horner's algorithm A3.2 five times, we have a time of

$$5(2t_{add} + 3t_{mult} + 4t_{mem} + 2t_{index}) + 4t_{comm} \quad (4.5)$$

to which we must add the time for extra memory accesses due to the various coefficients of the already computed polynomial in one variable not being in the memory of the correct processor for the computation of the polynomial in two variables as well as time for communication. Using the most efficient choices causes the total obtained by using A3.2 five times to change from equation (4.5) to

$$10t_{add} + 15t_{mult} + 28t_{mem} + 14t_{index} + 7t_{comm}. \quad (4.6)$$

A similar design using a parallel Knuth's algorithm A3.2a five times gives a total time of

$$5(2t_{add} + 2t_{mult} + 5t_{mem} + 2t_{index} + t_{comm}) + 8t_{mem} + 4t_{index} + t, \quad (4.7)$$

where t represents the time for "pre-processing" the polynomials in a single variable to become coefficients of a polynomial in two variables. This last step is necessary since the output of Knuth's method is a polynomial in standard, not pre-processed form. This pre-processing takes many additions and multiplications and cannot be ignored since it is done whenever we evaluate the polynomial in two variables. Thus this algorithm does not easily parallelize.

The other method that we consider involves using sequential methods as long as possible. Here

we use processor 1 to compute the polynomials Q_0 and Q_2 with processor 2 computing Q_1 and Q_3 followed by a parallel evaluation of the polynomial in two variables. The total time is

$$2(3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) + 6t_{mem} + 2t_{index} + \\ 3t_{add} + 3t_{mult} + 4t_{mem} + 2t_{index} + 8t_{mem} + 4t_{index} + t_{comm}$$

for a total of

$$9t_{add} + 9t_{mult} + 24t_{mem} + 10t_{index} + t_{comm} \quad (4.8)$$

which is considerably smaller than either (4.6) or (4.7). This proves (4.3). To prove (4.4), we simply apply the same reasoning to the case of four processors and keep track of added communication, memory and index times to obtain the desired result. This completes the proof of the theorem.

Theorem 3 : Let $P(u,v)$ be a bicubic polynomial in the real variables u and v and suppose that P is to be evaluated at N points without using matrix methods. Then the speedup in the evaluation time is asymptotically the same as in theorem 2 for the case of two processors and

$$\frac{4N(15t_{add} + 19t_{mult} + 45t_{mem} + 40t_{index} + 5t_{mem})}{(5N + 8)(3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) + 8t_{mem} + 4t_{index} + 4Nt_{comm}} \quad (4.9)$$

for four processors. If five processors are available, the speedup improves to

$$\frac{N(15t_{add} + 19t_{mult} + 45t_{mem} + 40t_{index} + 5t_{mem})}{(N + 1)(3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) + N(8t_{mem} + 4t_{index} + 4t_{comm})} \quad (4.10)$$

Proof.

For a two processor system, one of the processing elements will be idle much of the time in the final computation of the bicubic. This is still the case if there are many points at which the bicubic is to be evaluated and hence the speedup is essentially limited by the factors slowing down the evaluation at a single point. The maximum speedup is dependent on careful programming to minimize idle time; in any event it is similar to the results previously given.

Note that the effect of t_{mem} has been changed in the numerator of (4.9) and (4.10). This was done to account for the new loading of the values of the coefficients into memory at initialization; this was not considered previously. To evaluate a bicubic polynomial at N points using four processors, we must also have some amount of idle processor time. The results from each of the single variable evaluations must be made available to the processor performing the evaluation in the second variable. The

evaluations being performed are given below.

PROCESSOR 1	PROCESSOR 2	PROCESSOR 3	PROCESSOR 4
1	1	1	1
1	2	2	2
2	3	3	3
2	3	4	4
3	4	4	5
4	5	5	5
5	6	6	6

The state of available processors is precisely the same in this last state as it was in the second state where evaluation 2 was being done on processors 2, 3, and 4. Thus the time for evaluation of N points is the sum of an initialization time plus a time for getting to the same state plus a time for termination. The total is dependent on the value of N modulo 4; in the simplest case it is

$$(5N + 8)((3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) + 8t_{mem} + 4t_{index} + 4Nt_{comm})/4 .$$

Considerable improvement is possible if we have five processors available. In this case, the evaluations of the polynomials in a single variable are done on four processors and the fifth processor is devoted to evaluation of the bicubic after data is transmitted to it. The total time for evaluation of a bicubic at N points is thus approximately N times the time for evaluation of a single variable polynomial with an additional time added for evaluation of the last bicubic on the fifth processor. The actual total is

$$(N + 1)(3t_{add} + 3t_{mult} + 7t_{mem} + 3t_{index}) + N(8t_{mem} + 4t_{index} + 4t_{comm}).$$

This completes the proof of the theorem.

The next method we consider is that of using pipelined architectures. Modifying the method of deCasteljau leads to a pyramidal architecture scheme in which there are 16 nodes at lowest level, grouped into 4 collections of four nodes each. Each of these collections computes a different polynomial in one of the variables and sends its output to another collection of nodes using the same pipelined design as considered in the previous section. This last collection of nodes takes the values of the second independent variable and continues as in the deCasteljau model in the previous section to compute the values of the polynomial in terms of the second independent variable and thus produces the output of a polynomial of appropriate degree in both variables. As before, the polynomial is not in standard form but instead appears in a form conducive to numerical computation. The next result is easily obtained using the same reasoning as in the analysis of deCasteljau's method in the previous section; the proof is omitted. As before, efficiency larger than 1 is possible for this algorithm using a pipelined architecture.

Theorem 4: The speedup using deCasteljau's method instead of sequential evaluation to evaluate a bicubic polynomial at N points is

$$\frac{5(42Nt_{mem} + 12Nt_{mult} + 12Nt_{add})}{(N + 6)(t_{mem} + t_{mult} + t_{add})}$$

The efficiency is $60N/(N+6)$, which is greater than 1 if $N > 12$.

5. DISCUSSION

The primary motivation for the research in this paper was the controversy generated by the papers [2], [5], and [9] and the observation in [4] that many parallel algorithms do not perform at or even near their theoretical limits on many actual parallel machines. The algorithms were implemented in assembly language on a simulator which can be configured to model a number of different distributed system architectures and CPU's. Implementing these algorithms using low level assembly languages emulated the actual running of a computer and avoided making any assumptions about the quality of code generated by a compiler or assembler. Each load or store instruction was counted as a memory access and it was assumed that the time for data transfer into and from memory is the same. In general, the observed execution times showed good agreement with the theoretically obtained times described earlier

in the paper.

The optimal number of processors for parallel computation of second, third, or fourth degree polynomials of a single variable at a single input value on non-pipelined systems is 2. While it is possible to obtain a parallel algorithm for particular choices of polynomials, communication and synchronization constraints make it difficult to design an efficient generalized parallel algorithm which is independent of either underlying connection architecture or the actual architecture of the processor elements.

Note also that the models of computation described in this paper all assumed that the processors shared no common memory; this assumption is common in hypercube computers. However, shared memory simply reduces the time for inter-processor communication and thus the value of the quantity t_{comm} will be smaller than in a non-shared computer. Of course, the time represented by it will still be present in any real situation. For example, the reference [3] has an example of the use of the method of finite differences to speed up sequential evaluation of bicubics. Unfortunately, it seems that a pipelined method such as the method of finite differences can only be used on a shared memory multiprocessor system. No speedups were found using this method for evaluation of bicubics on our simulator for any architecture.

For quadratic functions, Horner's method provides the best pre-processing scheme. Hence the sequential processing of a quadratic will exploit this scheme. On the other hand, for $n \geq 3$, the scheme by Knuth [6] will be utilized. In this example also, there is no need for more than 2 processors. Even for $n = 4$, the optimal number of processors is apparently 2. Furthermore, parallelizing evaluation of cubics via Knuth's pre-processing scheme on a 2-processor computer cuts the multiplications to 2, additions to 2, and memory accesses to 5 with 1 inter-processor communication. Nevertheless, the speed-up obtained is actually an approximate value because of the pre-processing involved. To obtain the parameters in the pre-processing scheme, $n+1$ simultaneous equations have to be solved. The roots of these equations may even be complex. Therefore if the same polynomial is not to be evaluated at numerous x values then this method may not be superior to Horner's. Given this fact, our experiment indicates that using Knuth's scheme for low degree polynomials, evaluation can be implemented on a 2-processor

computer but that the efficiency is considerably less than 1.

The optimal number of processors for evaluation of bicubics at N points is a multiple of 5 assuming that the communication times between processors is constant. If the number of processors is a multiple of 5, then the results given in theorem 3 can simply be divided by an appropriate constant.

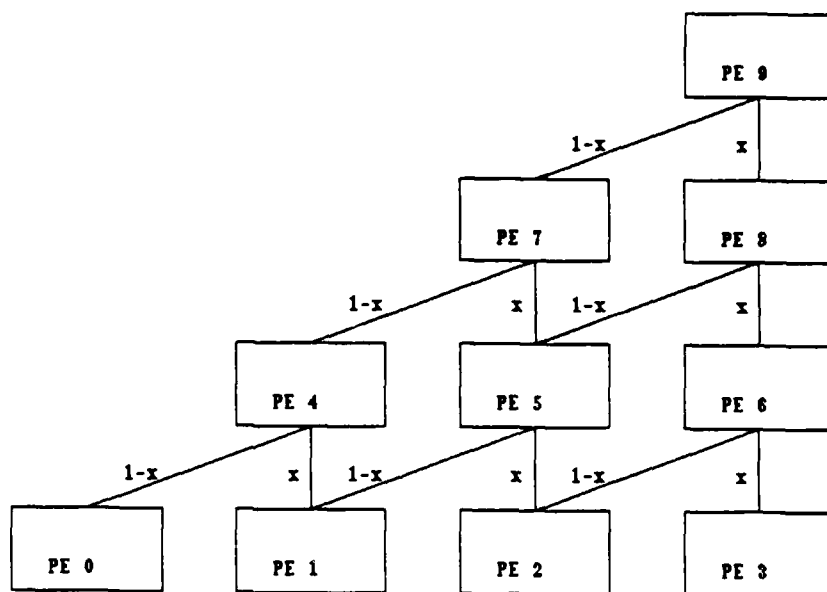
A final remark is in order about bottlenecks in a system for display of the bicubics that we considered in this paper. Most computer graphics systems use either a raster-type output device such as a CRT screen or laser printer in which the picture is scanned one line at a time or a vector device where a writing device moves from one point to the next rather than a line at a time. The most common example of a vector device is a plotter although vector CRT screens are still available. In general, all of these devices are sequential and vector devices in particular are not well suited to parallelization. Perhaps the next step is special parallel output devices.

Acknowledgement

Research of Ronald J. Leach was partially supported by the U.S. Army Research Office under contract number DAAL-03-86-G-0085. Research of Ronald J. Leach, Razeyah R. Stephen, and O. Michael Atogi was supported by a Howard University Research Grant.

REFERENCES

1. W. S. Dorn, Generalizations of Horner's Rule for Polynomial Evaluation, *IBM J. Res. and Devel.* 6 (1962) 239 -245.
2. V. Faber, Olaf M. Lubeck and Andrew B. White, Jr., Comments on the Paper "Parallel Efficiency can be Greater than Unity", *Parallel Computing* 4 (1987) 209-210.
3. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., (1982).
4. J. J. Hack, Peak vs. Sustained Performance in Highly Concurrent Vector Machines, *IEEE Computer* 19 (1986) 11-19.
5. R. Janben, A Note on Superlinear Speedup, *Parallel Computing* 4 (1987) 211-213.
6. D. E. Knuth, *The Art of Computer Programming, vol 2, Semi-Numerical Algorithms*, Addison-Wesley, Reading Mass., (1969).
7. H. T. Kung, New Algorithms and lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences, *JACM* 23 (2), (1976), 252-261.
8. I. Munro and M. Paterson, Optimal Algorithms for Parallel Polynomial Evaluation, *J.Comp. and Sys. Scis.* 7 (1973), 189-198.
9. D. Parkinson, Parallel Efficiency can be Greater than Unity, *Parallel Computing* 3 (1986) 261-262.
10. S. Winograd, On the Number of Multiplications Necessary to Compute Certain Functions, *Comm Pure & Applied Math* 23 (1970) 165-179.



GEOMETRIC CONSIDERATIONS IN BLENDING SURFACES

Ronald J. Leach

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

1. INTRODUCTION

A major problem in solid modeling is computing a "blending surface" between two intersecting surfaces. In a typical example, the surfaces are described by some polynomial equation of low degree and the blending surface is intended to provide a transition between the surfaces. The actual form of the blending surface or its equation are generally not as important as the need to rapidly compute the surface in order to be able to use it in an interactive system. For an example of a blending surface, look at a standard telephone.

One obvious method of describing the blending of two surfaces is to use the surface that is swept out by a rolling ball tangent to both of the surfaces. Unfortunately, this method often has the undesirable effect of having blending surfaces which have much higher degree than the surfaces to be blended, which are often quadric. Strange geometric effects are often associated with this type of blend, particularly when there are more than one surface to be blended.

Many authors ([2],[3],[4],[6],[8],[9] and [10]) have considered the problem of the generation of blending surfaces. Middleditch & Sears [6] describe blending by a method which gives blending surfaces of degree 4 if the surfaces to be blended are of degree 2. In [10], the author considers using other blending surfaces described by toroids and cylindrical pieces in which the blending surfaces also have low algebraic degree.

Hopcroft & Hoffman have taken another view. In a collection of papers ([2], [3] and [4]), they develop a method that they call the projective method. This method uses some techniques from algebraic geometry to obtain a surface which blends two surfaces by choosing curves on each and then obtaining a surface which is tangent to the surfaces at the given curves. In [3], they show that if the curves are quadratic curves and the surfaces are quadric, then the blending surfaces are of degree four.

All of these methods lead to surfaces which blend the existing surfaces. In many but not all applications, they provide satisfactory results from an aesthetic point of view. We note that there are typically two concerns for blending surfaces in these papers: rapid computation and description of the surfaces by simple equations of relatively low degree. The equations of low degree are desirable since blending is often followed by shading and by ray tracing. The shading and ray tracing algorithms are far simpler and faster than if the degree of the surface is small. The only geometric constraint ever used is that of tangency to the surfaces, often in prescribed curves.

In this note, we consider the generation of blending surfaces which attempt to minimize the surface area of the blend. Our technique will involve a mathematical theory known as the calculus of variations. We present an analysis of the difficulties involved with this technique and show the application to several problems.

This paper is grouped into four sections. Section 1 is the introduction. In section 2, we give some results from the calculus of variations that are relevant to blending surfaces. In section 3, we apply these results to some simple blending surfaces. In section 4, we analyze the results obtained and

close with some suggestions for implementation and for future work.

2. RELEVANT NOTIONS FROM THE CALCULUS OF VARIATIONS

The calculus of variations was largely developed by Euler and Lagrange in the seventeenth century. It is concerned with the problem of obtaining a curve or surface which maximizes or minimizes some function of the curve or surface. In the case of a curve, the problem to be solved generally requires minimizing or maximizing the functional

$$J[y] = \int F(x, y, y')$$

where F is some function of x , the unknown curve $y = y(x)$ and y' represents the derivative of y with respect to x . For simplicity, the limits of integration have been omitted. The solution to this problem must satisfy the well-known Euler differential equation

$$F_y - \frac{d}{dx} F_{y'} = 0$$

These results can be extended to problems involving surfaces and higher derivatives; in fact, we will use the extension to functions of surfaces later. A huge number of papers have been written on this subject. The text [1] is an excellent reference for the theoretical background needed for a complete understanding of the subject as used here as well as derivations of the major formulas. We will be concerned with two uses of the calculus of variations with regard to the problem of finding the surface of minimum area with certain restrictions.

The first problem we consider is the following [1, p20-21]. Among all smooth curves which pass through two fixed points P and Q , find the one for which the area of the surface of revolution formed by revolving the curve about the x axis is a minimum. This leads us to the problem of finding a function $y = y(x)$ whose graph passes through P and Q and which minimizes the surface area given by the integral

$$J[y] = \int y \sqrt{1 + dy/dx^2} dx$$

In this case, the Euler differential equation becomes

$$F - y' F_{y'} = 0$$

which has the solution

$$y = C \cosh((x + K)/C)$$

This function y will be the unique solution of the problem provided that a single curve of this form can be drawn through P and Q . There is always a smooth solution provided that the slope of the line joining them is sufficiently small; in the limit as $x_2 - x_1$ approaches 0, the slope must approach 0. Note that the graph of the function $y(x)$ is a catenary which is the shape of a heavy cable with no extra load and which is acted only by the force of gravity. The surface of revolution is also well-known; it is called a catenoid.

In the next example, we consider the much more interesting problem of surfaces that are not surfaces of revolution [1, p22-24]. For the case where the surface can be written in the form $z = z(x, y)$, the problem is to find the minimum of the functional

$$J[y] = \iint \sqrt{1 + z_x^2 + z_y^2}$$

and thus Euler's equation has the form

$$r(1 + q^2) - 2spq + t(1 + p^2) = 0$$

where

$$p = z_x, q = z_y, r = z_{xx}, s = z_{xy}, t = z_{yy}$$

The left hand side of Euler's equation is the numerator in the equation of the mean curvature of the surface and thus we know that any surface which minimizes the surface area must have 0 mean curvature. See [1] and [5] for a discussion of the meaning of these terms in differential geometry. Note that this analysis did not make any use of the underlying region of integration and that in particular did not use the boundary conditions which are necessary for a blending surface. (For a discussion of this topic, see [1, p173-176]). In general, the non-linear partial differential equation is difficult to solve explicitly, even in the simplest cases. In addition, there are often many solutions to the partial differential equation for given boundary conditions which are not even local solutions to the problem of minimizing the total surface area.

We have two choices in this situation. We can ask for a complete solution of the partial differential equation with the appropriate boundary conditions and require that the solution also minimize the area. Of course, we must expect that a large amount of numerical computation will be necessary. Another choice is to find simple solutions to the Euler equation, even though we ignore the boundary requirements. This method will provide blending surfaces with 0 mean curvature, but is unlikely to actually minimize the surface area.

In this paper, we elect the second option. We wish to use as our solution of Euler's equation the following surface called Enneper's surface by Rassias [8, p429]. We will change the notation of [8] slightly to write the surface as a set in x-y-z space with the surface being the image of the disk $u^2 + v^2 < r$ in the u-v plane. The surface is given by the equations

$$\begin{aligned}x &= u + uv^2 - \frac{1}{3}u^3 \\y &= -v - u^2v + \frac{1}{3}v^3 \\z &= u^2 - v^2\end{aligned}$$

As is clear from figure 1, the surface is simply connected as a subset of R^3 and is simple (not self-intersecting) for $r < \sqrt{3}$ [8]. Consequently, it is simple for the unit square, which is a smaller set (figure 2). It is easy to check that this surface satisfies the Euler equation and thus has 0 mean curvature.

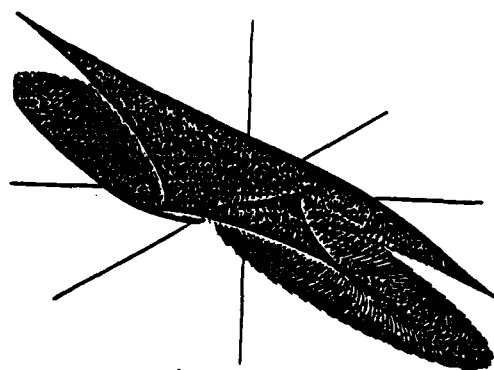


Figure 1

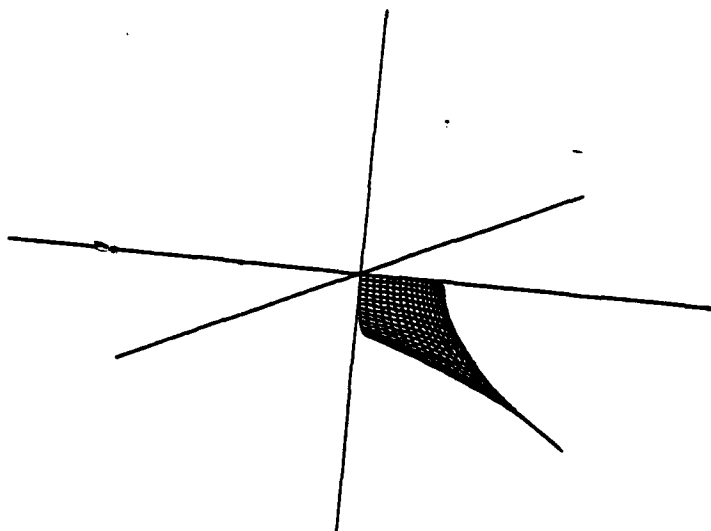


Figure 2

3. APPLICATIONS

In the previous section, we considered the problem of determining minimal surfaces with certain restrictions. We obtained a complete solutions only for surfaces of revolution. For other surfaces, we were able to obtain the restriction of the surface having 0 mean curvature and gave an example of a simple surface with this property. In this section, we apply these results to a few examples of simple blending surfaces.

Our first example is that of a cylinder of revolution intersecting a plane surface in a right angle. In this situation, we have surfaces of revolution and hence we may use the catenoid surface of the previous section. In figure 3 we show the two surfaces without blending and in figure 4 we use a blending surface. The blend is a catenoid obtained by rotating the curve

$$y = C \cosh((x + K)/C)$$

about the x-axis. The local coordinates $(-K, C, 0)$ are chosen for the intersection with the plane $x = 0$ and the coordinates $(0, C, 0)$ are chosen for the intersection with the cylinder.

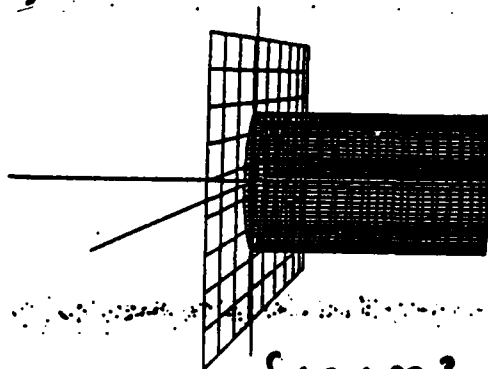


figure 3

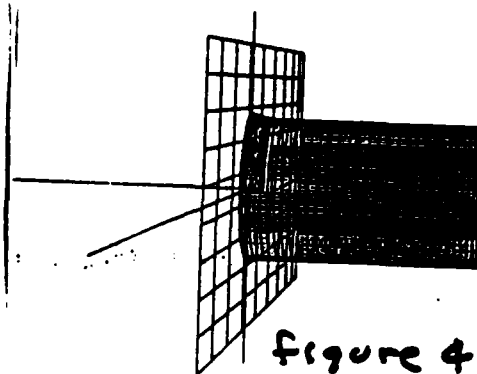


figure 4

Our second example is typical of many problems in this field in that simple geometric surfaces may lead to somewhat complicated blending surfaces. We consider the case of two right cylinders of equal radius intersecting at a right angle. In figure 5, the two cylinders are sketched and their intersection is given in bold. In figure 6, we use a blend consisting of portions of Enneper's surface.

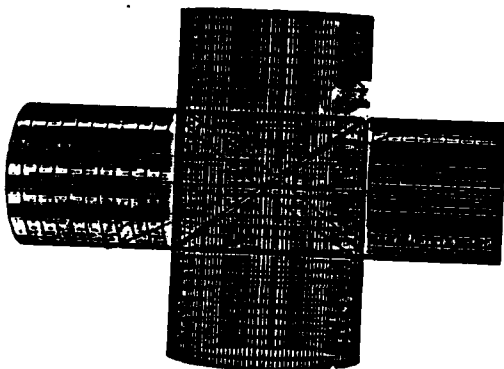


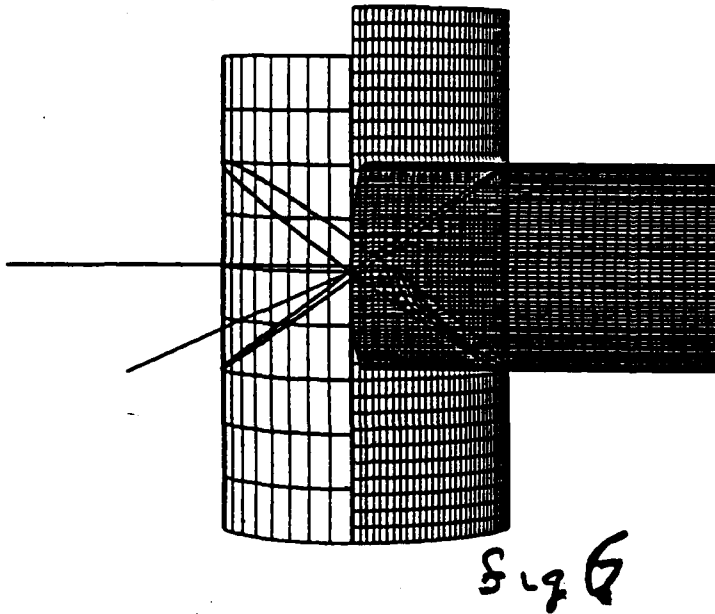
figure 5

4. ANALYSIS OF RESULTS OBTAINED

We have presented two surfaces to be used in blending problems. The primary goal was to minimize the surface area of the blending surface with an easily computed surface. Use of these blending surfaces have been demonstrated in two examples of somewhat different nature. It is reasonable to ask how these surfaces met the usual criteria for blending surfaces. That is, are they small perturbations, easily computed, geometrically significant, and well-suited to control by a user of a solid modeling system?

We first consider the question of speed. Using the cubic surface of Enneper described in this paper requires the computation of a cubic polynomial in two parameters u and v . For the blending of two quadratic surfaces, this is slightly faster than the computation of the general fourth degree blending surface that is generated by the projective method of Hoffman & Hopcroft or the fourth degree surface generated by the method of Middleditch & Sears. It is certainly faster than the higher degree surfaces that have been used by other authors. For higher degree surfaces, our method still uses a cubic blending surface while these other methods produce higher degree blends. In addition, our method works for non-polynomial surfaces whereas the projective method does not.

For situations such as in the first example considered in this paper, blending surfaces of revolution are useful. At first glance, they appear to be highly complex surfaces which require extensive computation to plot. However, the catenoid given here depends on two parameters C and K which depend on the points chosen as controls. We could have used a lookup table for the values of the catenary which is to be rotated. This speeds up the computation of the blending surface but may cause a large time penalty if ray tracing is to follow the blend because of the time needed to compute the intersections between the surface and rays of light. This excess computation may be reduced by replacing the hyperbolic cosine by the first few terms of its Taylor expansion about the origin as in figure 7. We note the similarity between the surfaces in figure 4 and figure 7. This is an example of another phenomenon in computer graphics in which several different approximations of a surface give similar pictures.



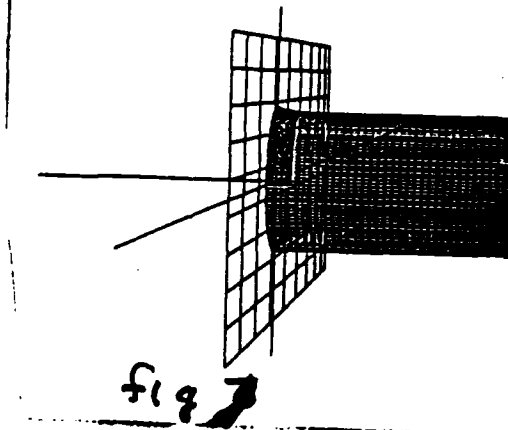
In the more difficult case where we do not have surfaces of revolution, the more general type of blending surface is needed. Here we use a cubic surface which is simple to compute. It often requires

a certain amount of patching but this is unavoidable in many instances.

In each of the situations discussed in this paper, we obtained blending surfaces that were small perturbations, easily computed and well suited to control by a user of a geometric modeling system. In the first case of surfaces of revolution blending into another surface of revolution about the same axis, the result minimizes surface area. In the second, more general situation, the proposed blend is a solution of Euler's equation but need not actually minimize the surface area.

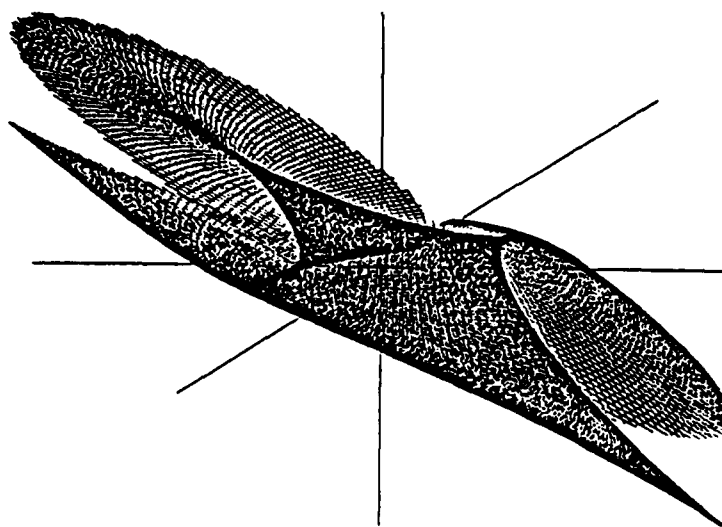
Unfortunately, these blending surfaces are not tangent to the given surfaces in general. We may elect to either accept these non-tangential blending surfaces as is or to reblend them near the points of intersection of the blending surface and the given surface. A "reblend" in this case would involve treating the blending surface as a small perturbation and then multiplying the values of the blend by a small parameter approaching 0 rapidly. Perhaps the use of lookup tables for the exponential function and the catenary for small values of the variable x would be appropriate in this context.

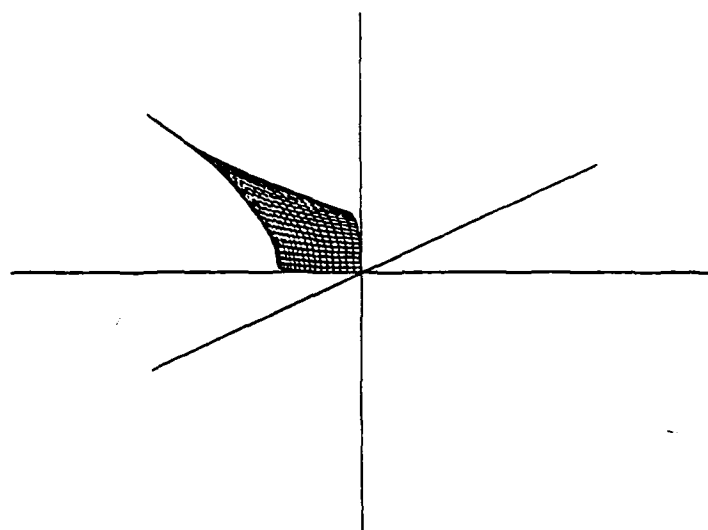
An additional problem with most existing approaches to blending surfaces is that the geometry of such surfaces is not always well understood or related to the "reality" of the situation. In particular, many blending algorithms when applied to even simple examples such as quadric surfaces which describe solid objects, cannot be guaranteed to lie entirely outside of the existing objects in the case of external blends. A similar problem also occurs for internal blends. Future work will consider this problem as well as the nature of the reblends mentioned earlier.

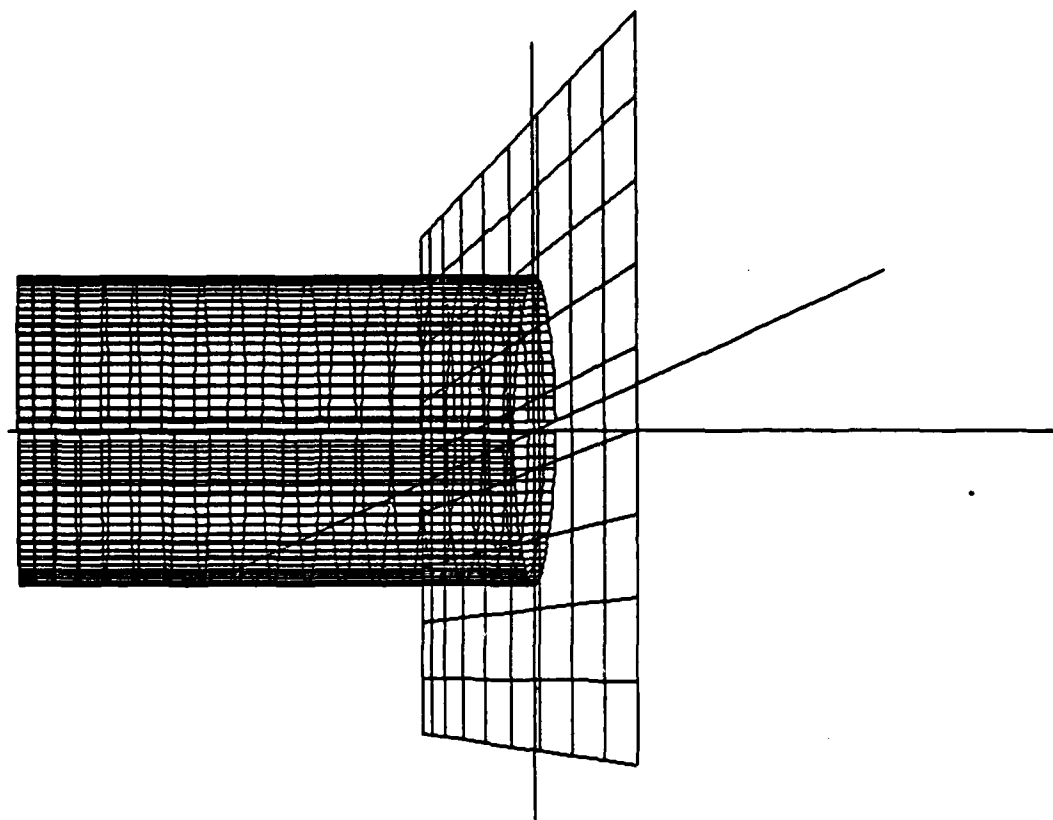


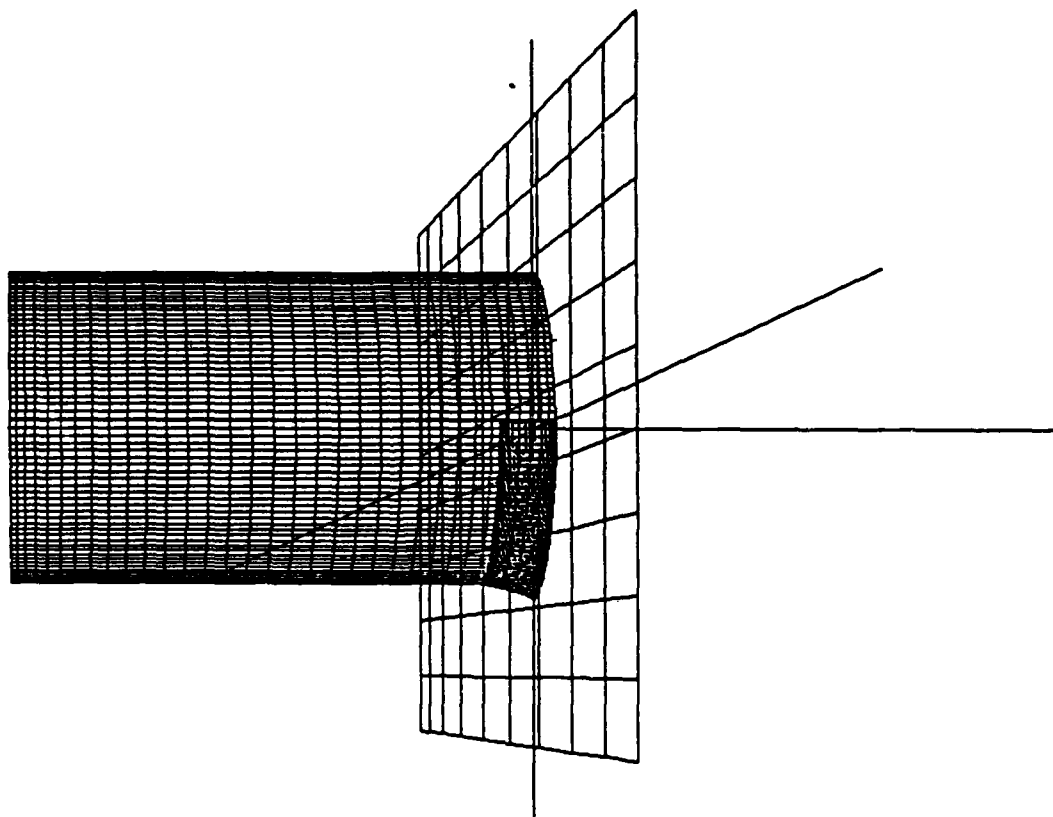
REFERENCES

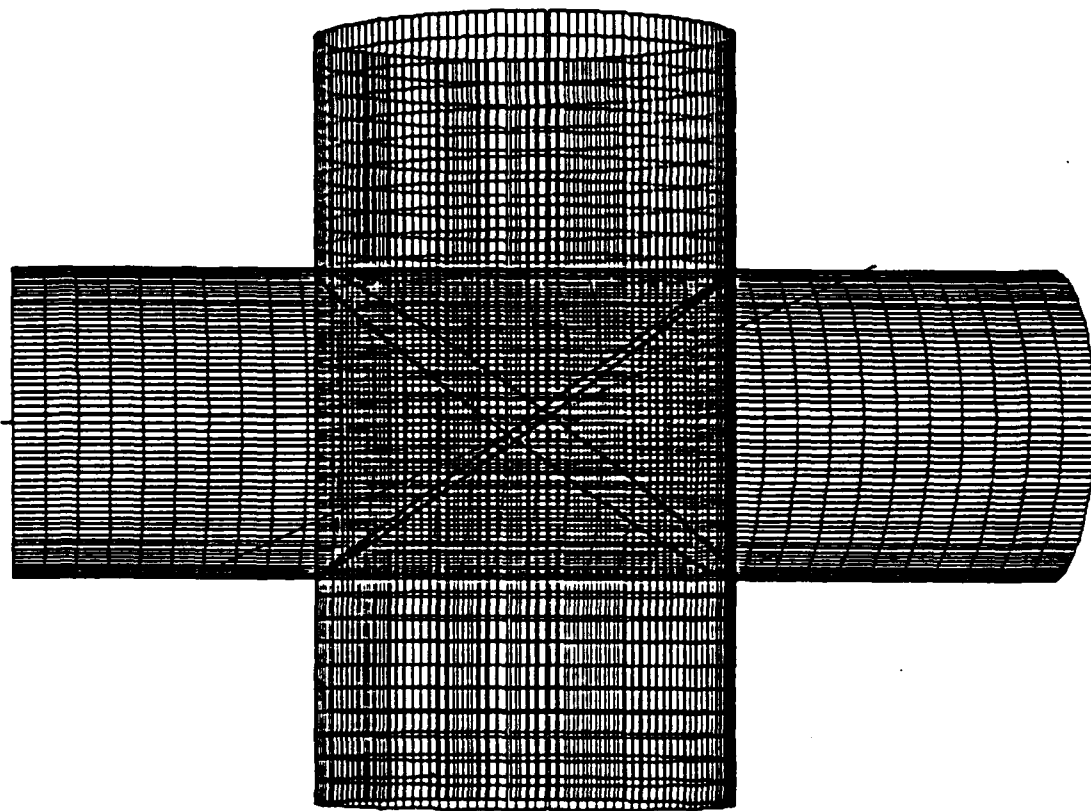
1. Gelfand, I.M., and S.V. Fomin *Calculus of Variations* (R.A. Silverman,trans.) ,Prentice-Hall Englewood Cliffs, NJ ,1963.
2. Hoffman, C.M. and J.E. Hopcroft, *Quadratic Blending Surfaces* , Computer Aided Design , 8,No. 6 , 301-306 , July/August 1986.
3. Hoffman, C.M. and J.E. Hopcroft, *Automatic Surface Generation in Computer Aided Design* , The Visual Computer , 1,No.2 , 92-100 , 1985.
4. Hoffman, C.M. and J.E. Hopcroft, *The Potential Method for Blending Surfaces and Corners* , Geometric Modelling (G. Farin,ed.) , SIAM , Philadelphia, PA, USA , 1986.
5. Holmstrom, L., *Piecewise Quadratic Blending of Implicitly Defined Surfaces*, SIAM Conference on Applied Geometry, Albany, NY, 1987.
6. Middleditch, A. and K. Sears, *Blend Surfaces for set Theoretic Volume Modelling Systems* , SIGGRAPH Comp. Graphics , 19 , 161-170 , July 1985.
7. Phillips, M. B., and Odell, G. M., *An Algorithm for Locating and Displaying the Intersection of two Arbitrary Surfaces* , IEEE CG&A , 48-58 , IEEE , Sept. 1984.
8. Rassias, Th., *On the Morse-Smale Index Theorem for Minimal Surfaces* , in *Differential Geometry, Calculus of Variations and Their Applications* , , 429-452 , Marcel Dekker , New York , 1985.
9. Rockwood, A., and J. Owen, *Blending Surfaces in Solid Geometric Modelling* , Proc. SIAM Conference on Geometric Modelling and Robotics , Albany,NY,USA , July 1985.
10. Rossignac,J., and A. Requicha, *Constant-radius Blending in Solid Modelling* , Comput. Mech. Engr. , 65-73 , July 1984.

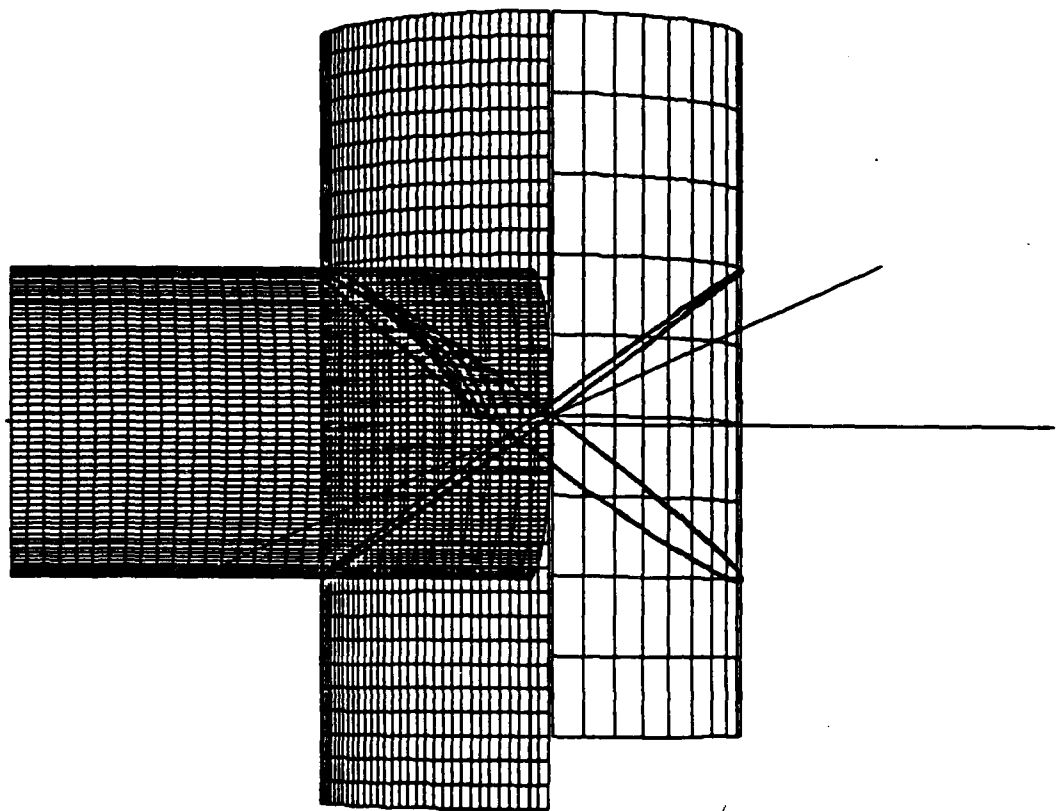


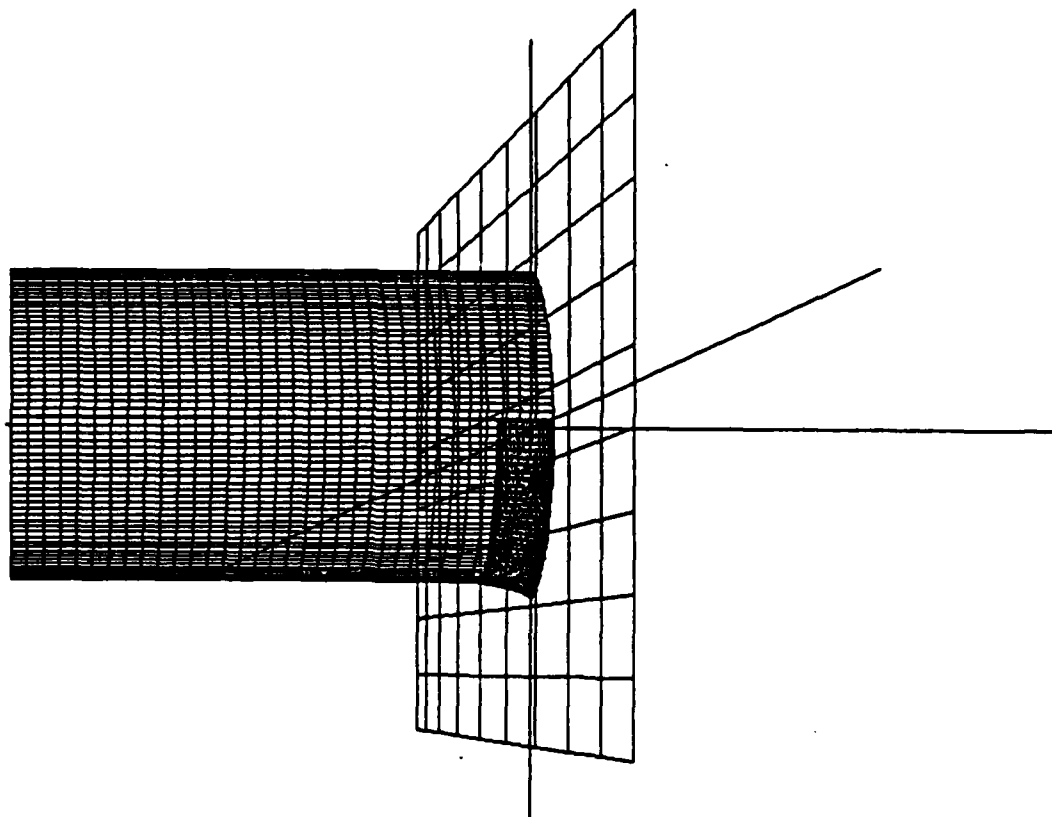












MINIMAL BLENDING SURFACES

Ronald J. Leach

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

1. INTRODUCTION

A major problem in solid modeling systems occurs when two solids intersect. Generally, it is not sufficient to simply compute the intersection of the two surfaces that bound the solids. It is often more important to replace the two surfaces near their intersection by a blending surface which allows a smooth transition between the surfaces. Blending surfaces should have the properties of being : easily computed, well suited to being accessed by other software in the solid modeling system, and being geometrically accurate. Many papers have been written on the subject of blending surfaces using a variety of techniques; [4], [5], [10], [11], and [12] are somewhat typical.

Some of these papers consider the blending surface as the final step in a representation and display process. They emphasize the aesthetic appeal of the blending surface and are somewhat less concerned with the analytic and computational properties of the blend. The blend surfaces obtained are typically appealing, but are often difficult to integrate rapidly into the other portions of the solid modeling system because of the high algebraic degree of the surfaces obtained. Other papers (e.g. [11]) emphasize the integration into a solid modeling system but do not emphasize geometric aspects.

Other techniques for blending surfaces emphasize the analytic requirements for rapid computation while not emphasizing the global geometric properties of the surface. An example of this is the projective method of Hopcroft and Hoffman [4] which computes the surface of minimal degree which is tangent to certain surfaces in prescribed curves. Their method gives the blending surface of minimal degree which is tangent to the given surfaces in the prescribed curves, but no conditions on the geometry of the surfaces is given.

The purpose of this note is to study the applicability of blending surfaces which are computationally tractable and which are also "minimal surfaces". Minimal surfaces have a particular property which is related to but not always identical with minimizing surface area; the exact relationship and the mathematical foundation is discussed in section 2. This minimizing property is very desirable in a solid modeling system since the goal of blending surfaces in such a system is to provide a smooth transition between solid objects which can be efficiently implemented by automated manufacturing tools such as milling systems.

In this context, computationally tractable generally means that the surface is of low degree. In this paper, we will consider those surfaces which are both computationally tractable and geometrically significant.

The paper is organized as follows. In the next section, the background results and terminology for the discussion of minimal surfaces is given. In section 3, we consider surfaces of degree 2 and show that the only minimal surfaces of degree 2 are planes. It would be natural to perform the same analysis for surfaces of degree 3 and 4 by considering all of the possible cases. However, there are 99 possible types of surfaces and many of these types involve so many coefficients that analysis is beyond the limits of the symbolic computation program MACSYMA (and hence beyond the limits of any human performing the algebraic computations). Thus in section 4 we consider an alternate representation for minimal surfaces. In section 5 we show some examples of the use of minimal blending surfaces and the effects of certain degrees of freedom on the behavior of the surfaces. Section 6 provides a summary of results. The paper closes with a discussion of some open problems.

2. MINIMAL SURFACES

A minimal surface is defined as the set of all points (x, y, z) which satisfy the second order partial differential equation

$$r(1 + q^2) - 2spq + t(1 + p^2) = 0 \quad (2.1)$$

where

$$p = z_x, q = z_y, r = z_{xx}, s = z_{xy}, t = z_{yy}. \quad (2.2)$$

In differential geometry ([1], [2], [7], [9]), it is well-known that the vanishing of the left hand side of this equation implies that the mean curvature of the surface is 0.

The partial differential equation is obtained as the solution to a "variational problem" of minimizing the surface area. An infinite number of functions satisfy the equation; this is not surprising since no boundary conditions are given. Unfortunately, even when complete boundary conditions are given in the form of requiring that the surface must pass through some boundary curve, there may be many solutions, not all of which actually minimize the surface area over the class of all curves which pass through the given boundary curve. The reader should think of soap bubbles as models of surfaces which minimize surface area.

A word on the methodology used in this paper is in order. As was indicated in this section, extensive use is made of the symbolic manipulation program MACSYMA to actually compute the low degree surfaces which are also minimal surfaces. MACSYMA is a trademark of Symbolics, Inc. The computations were performed on a SUN 2/120 workstation running UNIX. The workstation had a physical memory of 4MB and a virtual memory of approximately 20.1MB. The original goal of this research was to characterize all of the minimal surfaces which were of low degree by making use of MACSYMA for the computations. We will see in the next section that extensive analyses and simplifications had to be done in order to be able to use MACSYMA efficiently and not exceed the limitations of the workstation.

Of course, the limitations of MACSYMA and the memory of the workstation are relevant only during the characterization of those surfaces which are actually minimal surfaces. The actual computation of appropriate minimal blending surfaces in applications is quite rapid, once the preliminary analyses have been made.

3. SURFACES OF DEGREE 2

The most general surface of degree 2 is given by the implicit equation

$$F(x, y, z) = Ax^2 + Bxy + Cy^2 + Dxz + Eyz + Fz^2 + Gx + Hy + Iz + J = 0 \quad (3.1)$$

This equation involves 10 constants of which 9 may be arbitrarily chosen.

The goal of this paper is to characterize the low degree minimal blending surfaces. A brute force computation of the result of the differential operator of formula (2.1) on $F(x, y, z)$ using implicit differentiation leads to an expression which requires six pages to print and which cannot be factored directly within the memory limitations of the computer. Clearly a simplification was needed, especially, since the intention of this research was to classify the potential minimal blending surfaces of low degree. The only possibility is to consider classification of the various surfaces that can arise from the expression $F(x, y, z)$. One obvious simplification is to observe that all of the linear terms can be eliminated by translation. It is less clear that all of the mixed terms can be eliminated by appropriate rotations; the original observation of this was apparently due to Euler. We note that translations and rotations leave the minimal surface equation invariant. We will use the invariance of the minimal equation frequently without mention while considering possible simplifications of various expressions.

Removal of the mixed terms leaves several possibilities: the variable z does not appear and thus the surface is of the form

$$Ax^2 + By^2 = C \quad (3.2)$$

which is an elliptic cylinder, hyperbolic cylinder, or two planes; the variable z appears to only the first

power and thus the surface has the equation

$$z = Ax^2 + By^2 + C \quad (3.3)$$

which is either a paraboloid (elliptic or hyperbolic) or a parabolic cylinder; or the variable z appears to the second power and the surface has the equation

$$\frac{z^2}{c^2} \pm \frac{x^2}{a^2} \pm \frac{y^2}{b^2} = 1. \quad (3.4)$$

Here all of the unnecessary linear terms have been eliminated; none of a , b and c are 0. We consider separately the cases where z does not appear, where it appears to the first power and where it appears to the second power.

If z does not appear as in equation (3.2), then the minimal surface equation implies that the surface must be a portion of a plane.

If z appears to the first power only as in equation (3.3), then the minimal surface equation yields

$$2A(4B^2y^2 + 1) + 2B(4A^2x^2 + 1) = 0$$

which is only possible for arbitrary x , y , and z only if $A = B = 0$ and therefore the surface is a plane.

When the variable z appears to the second power as in equation (3.5), the minimal surface equation yields two possible solutions: either the surface is a horizontal plane or else

$$z = \frac{(E-D)Cx^2 + DC^3 + D^2C}{DC^2 - DE} \quad (3.6)$$

The first equation is that of a plane. The second equation (3.6) implies a restriction on the surface in that its points must also lie on another surface, which as we saw earlier was a paraboloid.

Thus we have obtained the following result.

Theorem

The only minimal surfaces of degree 2 are planes.

4. HIGHER DEGREE SURFACES

The most general surface of degree 3 is given by the implicit equation $G(x, y, z) = 0$, where

$$G(x, y, z) = Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2z + Fxz^2 + Gz^3 + Hy^2z + Iyz^2 + F(x, y, z) \quad (4.1)$$

and $F(x, y, z)$ is an arbitrary second degree expression in x , y , and z . As before, the minimal equation for this general expression is too complicated to submit to a symbolic algebra program and thus we resort to some simplifications. The number of possible cases is considerably larger than the number of possibilities for degree 2 surfaces. Salmon [13] indicates 23 possible forms which are classified according to what he calls their "class" and "singularities". An examination of 5 of these forms failed to yield any minimal surfaces after considerable computer time per example and hence an alternative approach was used. Note that the situation is even worse for surfaces of degree 4. Salmon indicates that there are 76 different species of surfaces. The references [3], [13], and [14] describe the possibilities for surfaces of degree 2, 3, and 4.

L. [2], the authors describe a method due to Weierstrass in which auxiliary mappings are used to find parametrizations of minimal surfaces. The method is:

1. Choose an open, connected subset of the plane.

2. Choose a complex-valued analytic function g and differential w in some domain D so that the expressions α_1 , α_2 , and α_3 defined in (4.4) - (4.6) satisfy

$$\sum \alpha_k^2 = 0 \quad (4.2)$$

$$\sum |\alpha_k|^2 > 0. \quad (4.3)$$

3. Form the expressions α_1 , α_2 , and α_3 by

$$\alpha_1 = (1 - g^2) \frac{w}{2}, \quad (4.4)$$

$$\alpha_2 = i(1 + g^2) \frac{w}{2}, \quad (4.5)$$

$$\alpha_3 = gw, \quad (4.6)$$

4. Integrate the expressions on the right hand sides of equations (4.4), (4.5), and (4.6) from 0 to z . Replace z by $u + iv$. To obtain the parametrization of the surface, set x , y , and z to be the real parts of the integrals.

It is easy to see that if each of the α_i is constant, then the surface is a plane. In addition, if g is constant, then the expressions in equations (4.4) and (4.6) are proportional to one another and hence the surface lies in a plane.

The choice $g = z$, $w = 1$ leads to the surface known as Enneper's surface [2] which is described by

$$x = u - \frac{u^3}{3} + uv^2, \quad (4.7a)$$

$$y = -v + \frac{v^3}{3} - u^2v, \quad (4.7b)$$

$$z = u^2 - v^2. \quad (4.7c)$$

This surface appears to be of high degree even though no terms in the parameterization have degree higher than 3; MACSYMA indicates a degree of 27 when using the "eliminate" command.

The choice $g = z$, $w = z$ leads to a surface described by the equations

$$x = \frac{u^2 - v^2}{4} - \frac{v^4 - 6u^2v^2 + u^4}{8},$$

$$y = -\frac{u^3v - uv^3}{2} - \frac{uv}{2},$$

$$z = \frac{u^3 - 3uv^2}{3}.$$

The result from a MACSYMA computation to eliminate the parameters and write in implicit form is even more complicated; it takes 1288 lines to display and has a degree of at least 32.

It is clear that minimal surfaces (except for planes) are either of high algebraic degree or not algebraic. Based on these observations, we simply present several well-known minimal surfaces by their parametric equations but will not attempt to write them in implicit form.

It seems likely from the above discussion that there are only two minimal surfaces of low parametric degree. However, each of these surfaces has additional degrees of freedom. The constant expression $w = 1$ can be replaced by the complex constant $e + fi$. The parameters e and f control the rotation and scaling of the surface. The effect of the other degrees of freedom is more striking. Each term linear in z can be replaced by a term of the form $A + Bz$, where A and B are complex; this provides four additional degrees of freedom in the surfaces. The effect of these additional degrees of freedom will be discussed in the next section.

The next three equations show how the original parametric equations of Enneper's surface (4.7a through 4.7c) are influenced by the extra degrees of freedom.

$$x = \frac{(1 - a^2 + b^2)u + 2abv}{2} - \frac{(ac - bd)(u^2 - v^2) - (ad + bc)(2uv)}{2} - \frac{c(u^3 - 3uv^2) - d(3u^2v - v^3)}{6} \quad (4.7d)$$

$$y = \frac{(1 + a^2 - b^2)v + 2abu}{2} - \frac{(ac - bd)2uv + (ad + bc)(u^2 - v^2)}{2} \quad (4.7e)$$

$$z = au - bv + \frac{c(3u^2v - v^3) + d(u^3 - 3uv^2)}{6} + \frac{c(u^2 - v^2) - 2duv}{2} \quad (4.7f)$$

This same possibility of increasing the degrees of freedom holds in all of the other surfaces considered in this section to an even greater degree.

The catenoid arises from the choices $g = e^z$, $w = -e^{-z}$, which leads to the parametrization

$$\begin{aligned} x &= \cos(v)\cos(u) - 1, \\ y &= \sin(v)\cosh(u), \\ z &= u. \end{aligned}$$

A minimal surface due to C. C. Chen [2] is given by $g = z + \frac{1}{z}$, $w = z^2$ and has the parametrization

$$\begin{aligned} x &= \frac{10u^3v^2 - 5uv^4 - u^5}{10} + \frac{3uv^2 - u^3}{6} - \frac{u}{2}, \\ y &= \frac{-v^5 + 10u^2v^3 - 5u^4v}{10} + \frac{v^3 - 3u^2v - v}{2}, \\ z &= \frac{u^4 - 6u^2v^2 + v^4}{4} + \frac{u^2 - v^2}{2}. \end{aligned}$$

The final surface that we present here is due to Jorge and Meeks [6]. It is obtained by choosing $g = z$ and $w = (z^{n+1} - 1)^{-2}$ and leads to the parametrization where the coordinates u , v , and z are the integrals of the real parts of the expressions α_i given by

$$\alpha_1 = \frac{1 - z^{2n}}{2(z^{n+1} - 1)^2}, \quad (4.8a)$$

$$\alpha_2 = i \frac{(1 + z^{2n})}{2(z^{n+1} - 1)^2}, \quad (4.8b)$$

$$\alpha_3 = \frac{z^{2n}}{(z^{n+1} - 1)^2}. \quad (4.8c)$$

5. ANALYSIS OF THE SURFACES

The goal of blending surfaces is to provide a transition between surfaces. A blending surface should be easy to generate, appear in a form which can be computed rapidly, and interface well with the solid modeling system in which it is being used. The surface should be aesthetically pleasing and represent only a small amount of additional material if the actual solid is to be created by use of a computer-guided milling machine. The use of the few fixed blending surfaces described in this paper certainly meets the criterion of being easy to generate since they are selected from a small list. They are presented here by their parametric representation and therefore can be computed rapidly, perhaps not even requiring the use of any patches. They share the same difficulty as any parametric representation in that intersections with other surfaces defined parametrically are much harder to compute than if the surfaces were described explicitly. They also allow a certain amount of possible interaction with a designer in the sense that some of parameters which provide the degrees of freedom can be altered to change the surface. We will illustrate this in detail for Enneper's surface which was discussed in the previous section; the parameters u and v will be restricted to the unit square. In [8], this surface was used for blending two intersecting cylinders of equal radius. The graph of Enneper's minimal surface is given in figure 1.

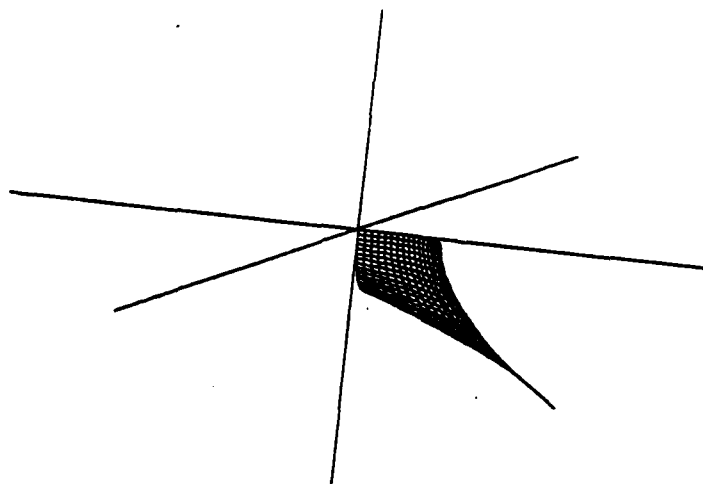


Figure 1 Enneper's surface

The effects of the parameters a , b , c , and d of formulas (4.7d, 4.7e, and 4.7f) on the graph of the surface are shown in figures 2-21. The figures are grouped by varying each of the parameters singly followed by consideration of several cases in which the parameters are allowed to vary together.

The first case that we consider is when the parameters b , c , and d are all 0 and the parameter a varies. In this case, the resulting surface obtained from the original Enneper surface is always a plane. The graphs are shown in figures 2-4.

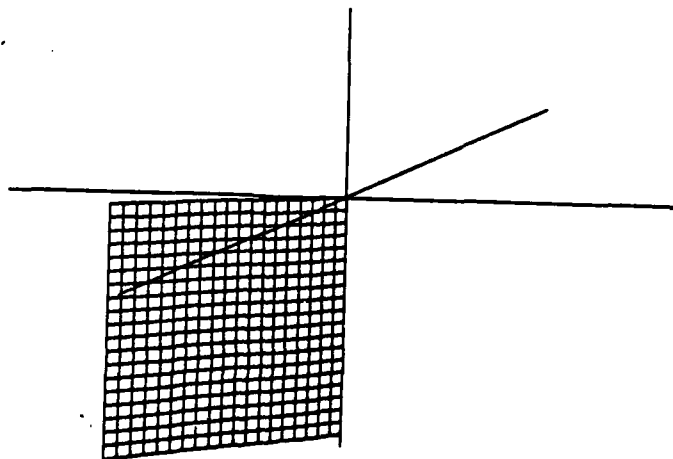


Figure 2 $a = 1, b=c=d=0$.

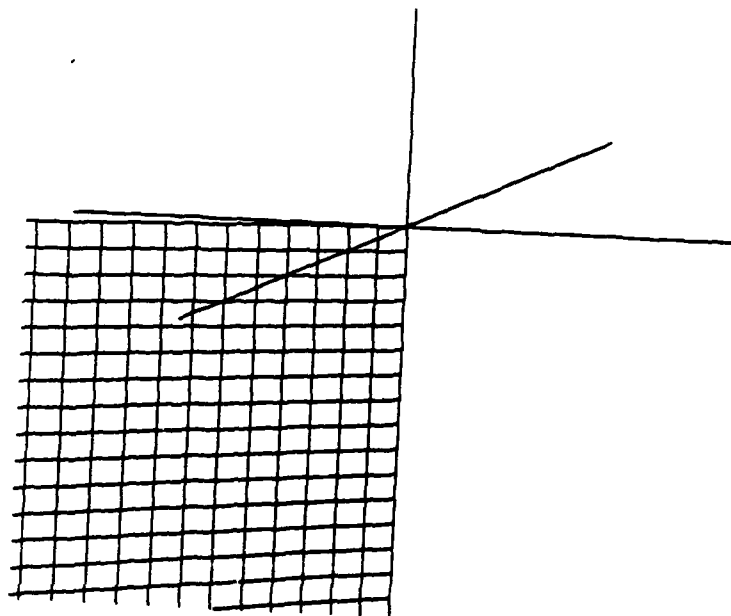


Figure 3 $a = 2, b=c=d=0.$

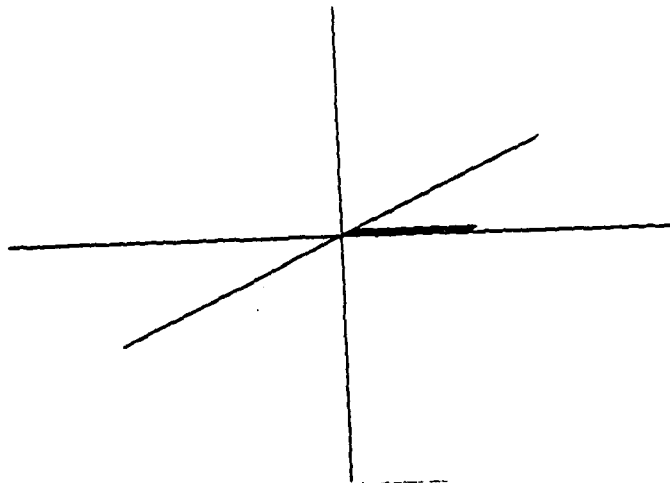


Figure 4 $a = 3, b=c=d=0$.

The effects of varying the parameters a, c , and d are shown in figures 5-6.

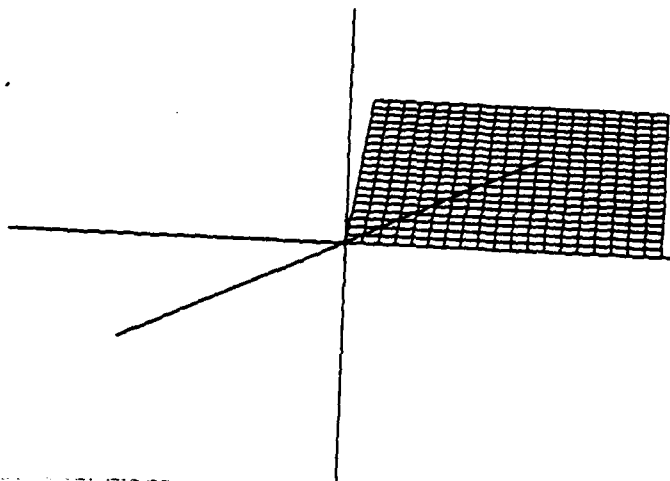


Figure 5 $b = 1, a=c=d=0$.

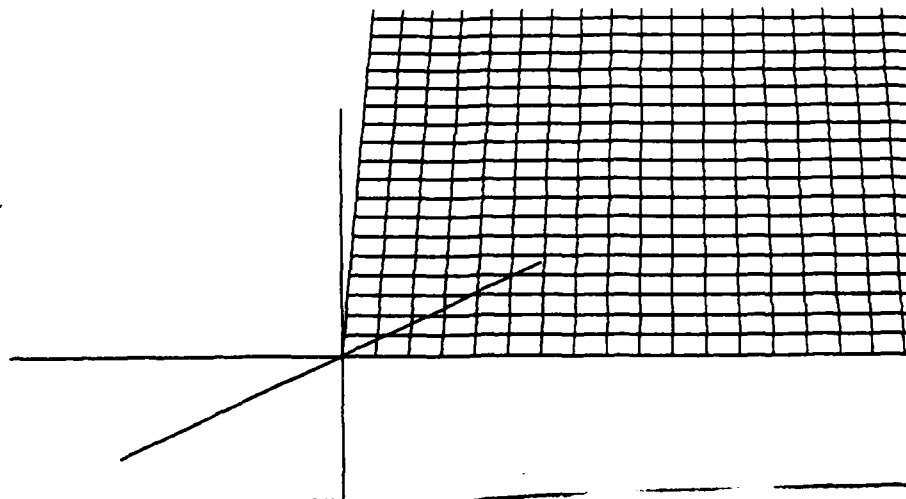


Figure 6 $b = 2, a=c=d=0.$

The effects of the parameter c on the graph is more interesting, at least in the range that we are showing here. The surface displays a twist which was not obvious in the original Enneper surface (the case $a = b = d = 0, c = 1$) which was shown in figure 1. The results are shown in figures 7-9.

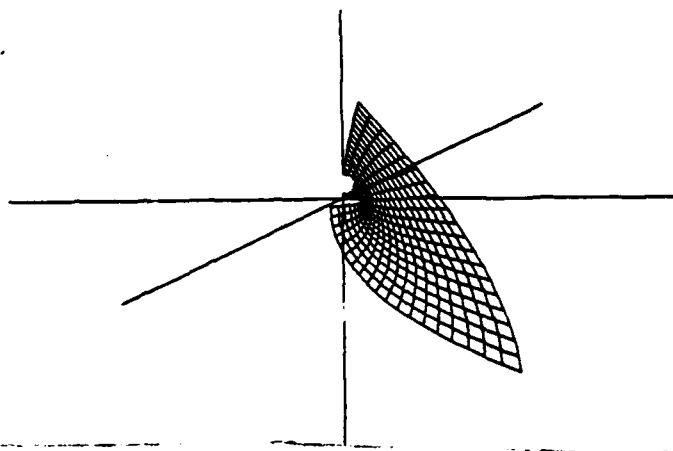


Figure 7 $c = 2, a=b=d=0.$

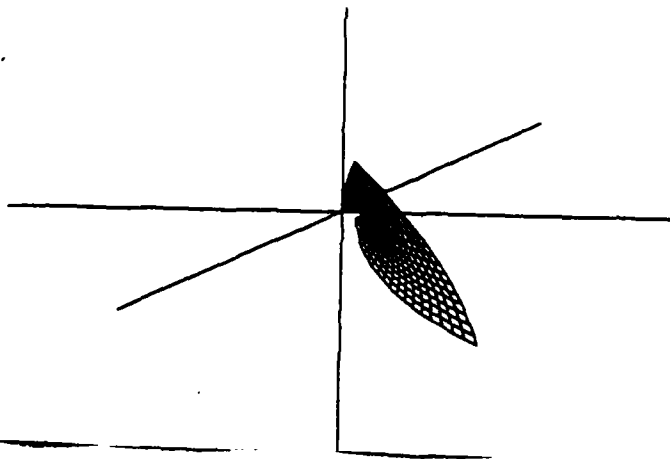


Figure 8 $c = 3, a=b=d=0$.

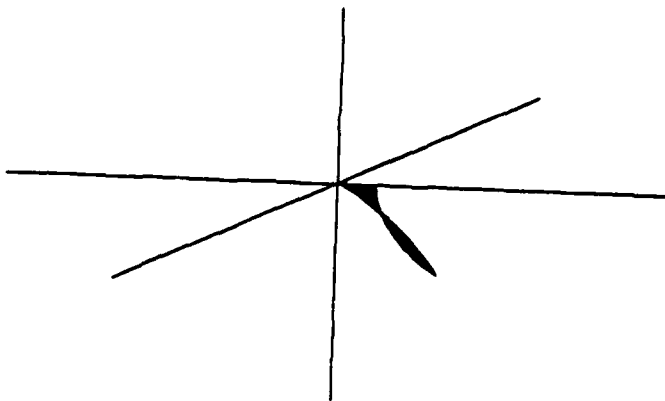


Figure 9 $c = 4, a=b=d=0$

The effect of the parameter d is shown in figures 10-12. Again there is a considerable twist in some of the surfaces.

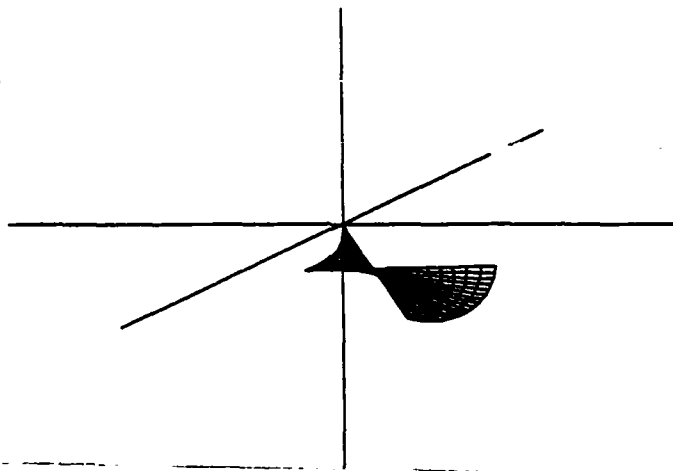


Figure 10 $d=1, a=b=c=0$.

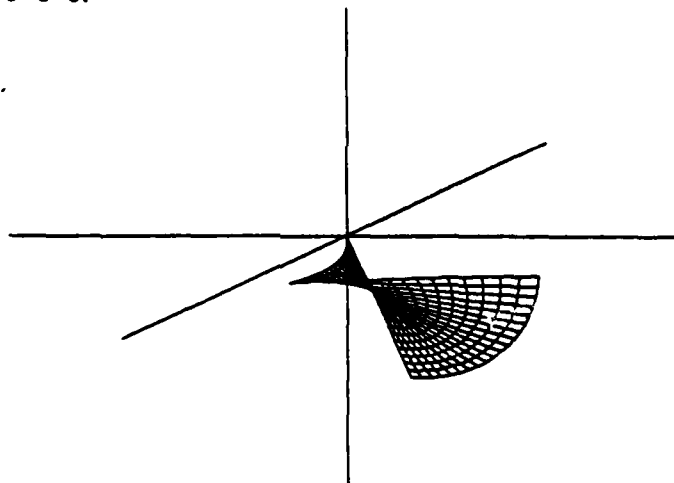


Figure 11 $d=2, a=b=c=0$.

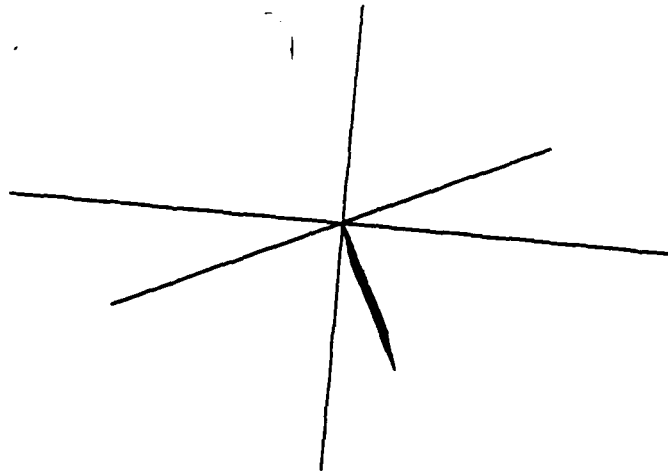


Figure 12 $d = 3, a=b=c=0$.

Figures 1-12 indicate the effect of changing the parameters one at a time. The next case to consider is when the parameters are varied two at a time. The simplest situation occurs when c and d are both 0. An examination of the effect of the parameters a and b on the parametric form of the surface given in equations (4.4), (4.5) and (4.6) for Enneper's surface shows that the vanishing of c and d implies that the surface is a plane. For this reason, we omit the graphs in this situation.

In figures 13-20, we show the result of some variations in the parameters a and c .

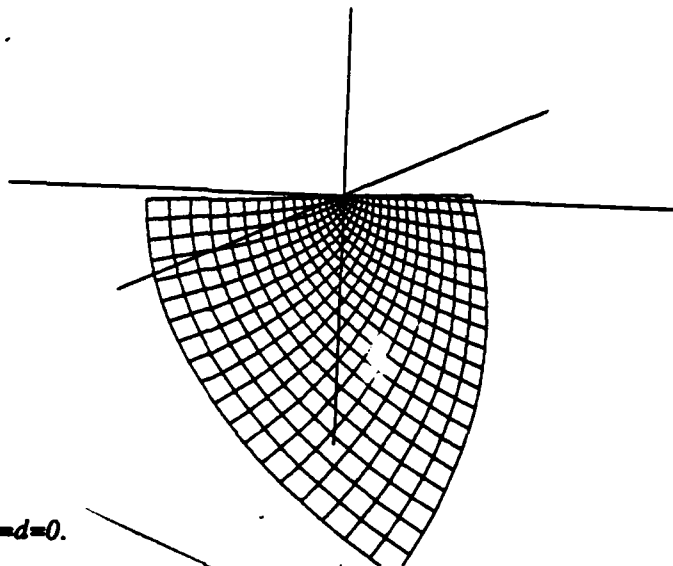


Figure 13 $a = 1, c = 1, b=d=0$.

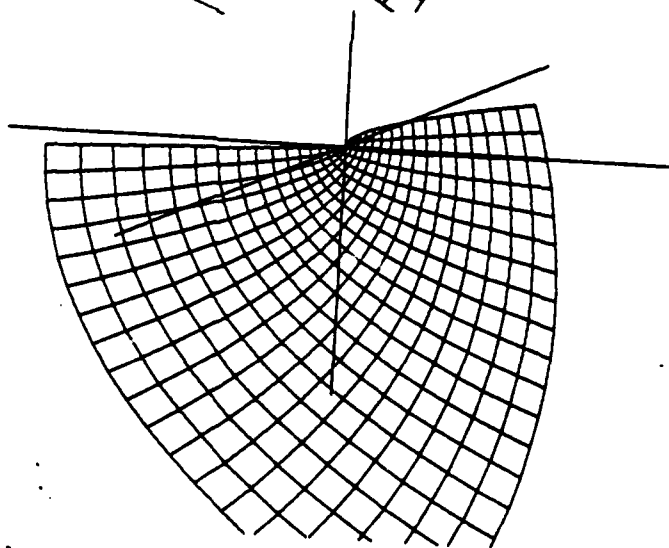


Figure 14 $a = 1, c = 2, b=d=0$.

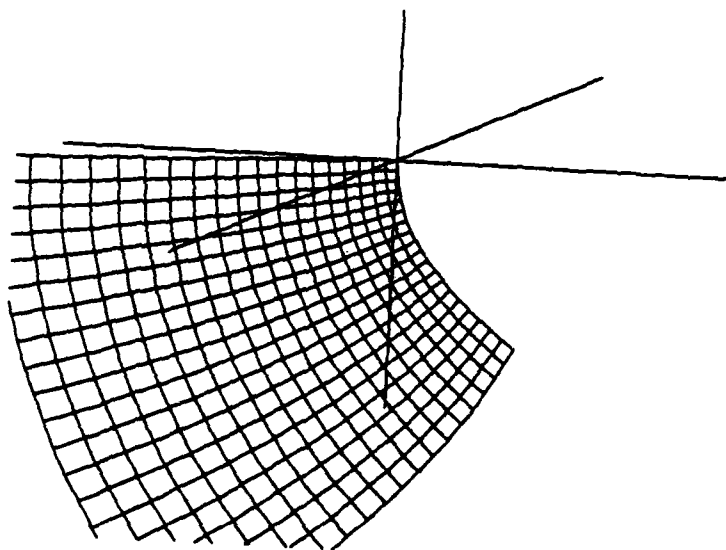


Figure 15 $a = 1, c = 3, b = d = 0$.

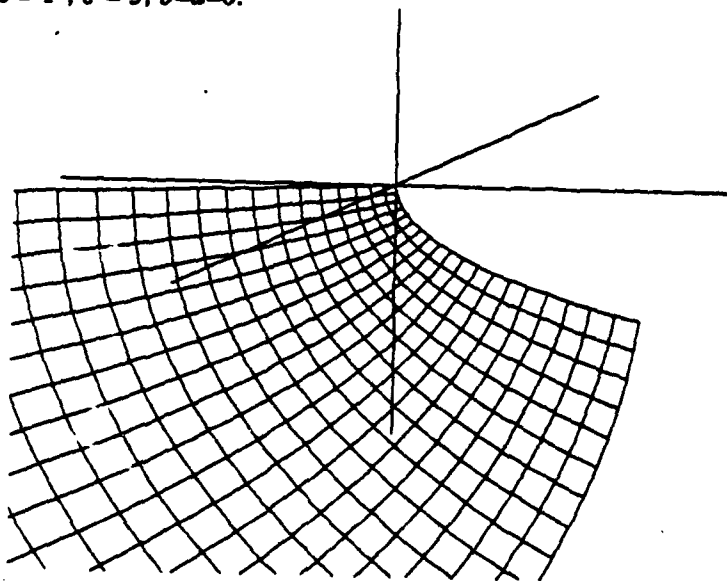


Figure 16 $a = 2, c = 1, b = d = 0$.

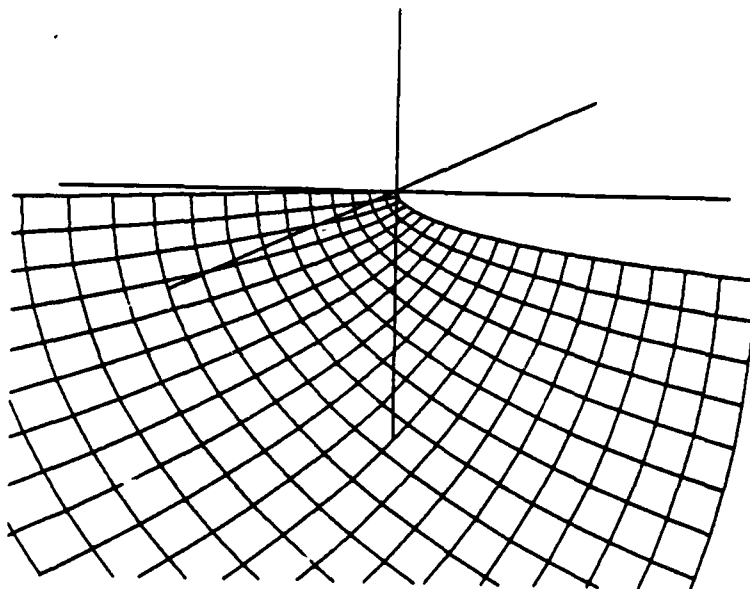


Figure 17 $a = 2, c = 2, b=d=0$.

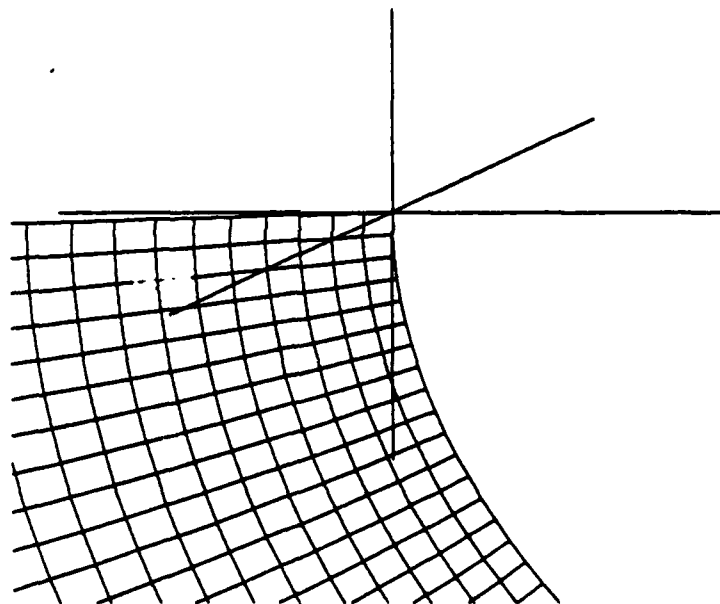


Figure 18 $a = 2, c = 3, b=d=0$.

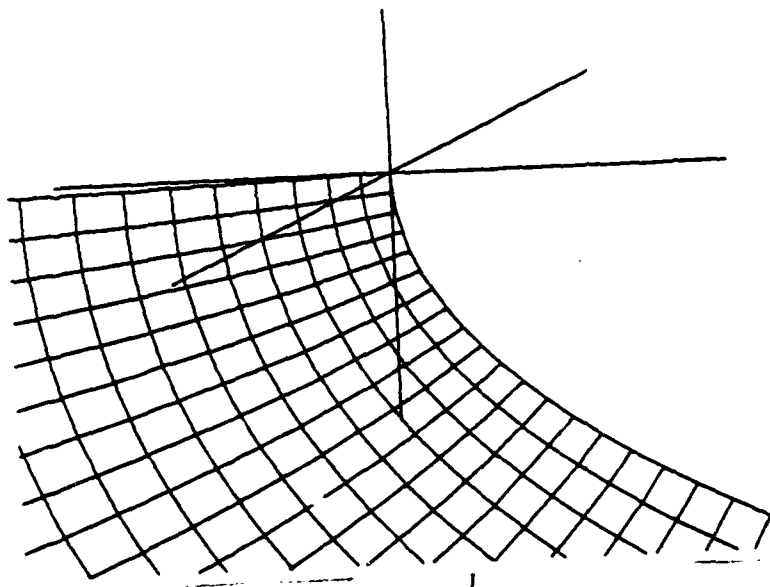


Figure 19 $a = 3, c = 1, b=d=0.$

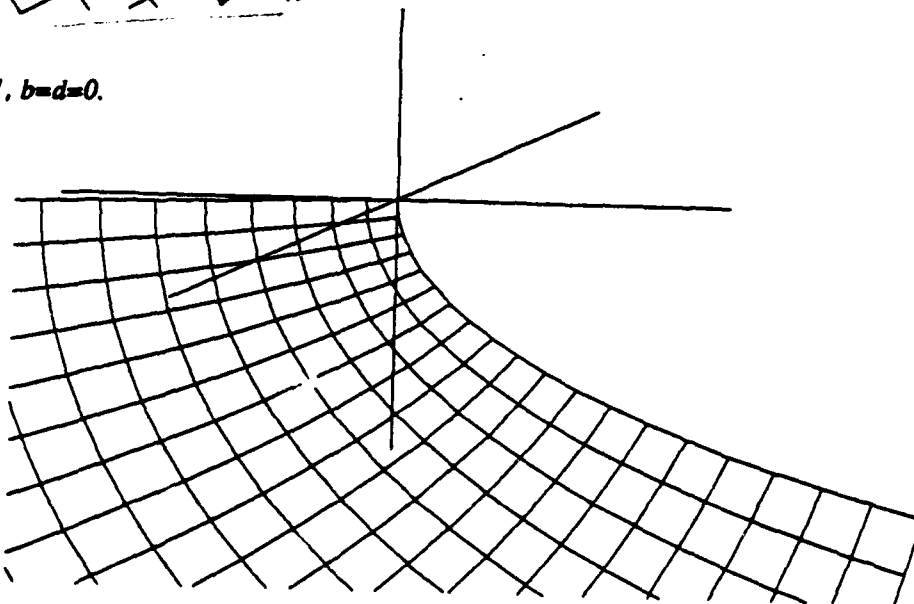


Figure 20 $a = 3, c = 2, b=d=0.$

Notice that the level curves in the upper left corner of the graph in figure 20 are nearly straight, indicating that this type of surface could be useful for blending where one of the surfaces is a plane.

The last variation of this surface that we give in this paper is shown in figure 21. It uses the values $a = 2$ and $c = 3$ with b and d being 0. Many of the other choices of a, b, c , and d in the range of integers 0 .. 4 lead to surfaces which have cusps and are therefore unsuitable for blending surfaces.

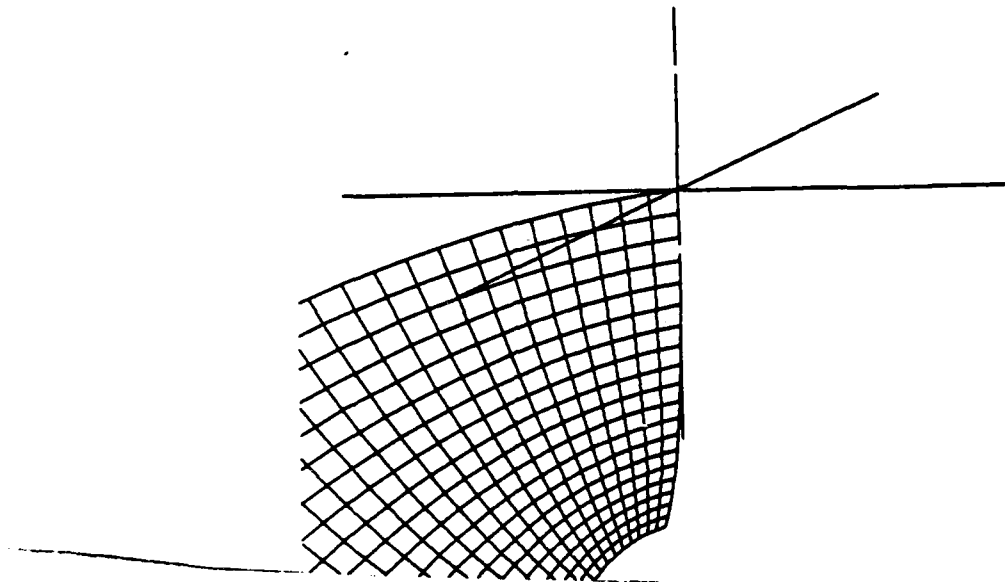


Figure 21 $a = 2, d = 3, b=c=0$.

The surfaces given in figures 1-21 indicate some of the modifications that can be made to a standard surface using certain degrees of freedom which can be described by the parameters a, b, c , and d . Additional degrees of freedom are available in the parameters e and f . These two parameters were not used here because in this example, they simply cause rotation and scaling of the surface.

The other potential blending surface that we show here is the Jorge-Meeks surface. This surface is useful in blending together n cylinders of equal radius which emanate from the same center. The graph of the surface is presented here only in the case $n = 3$. It originally appeared in [2].

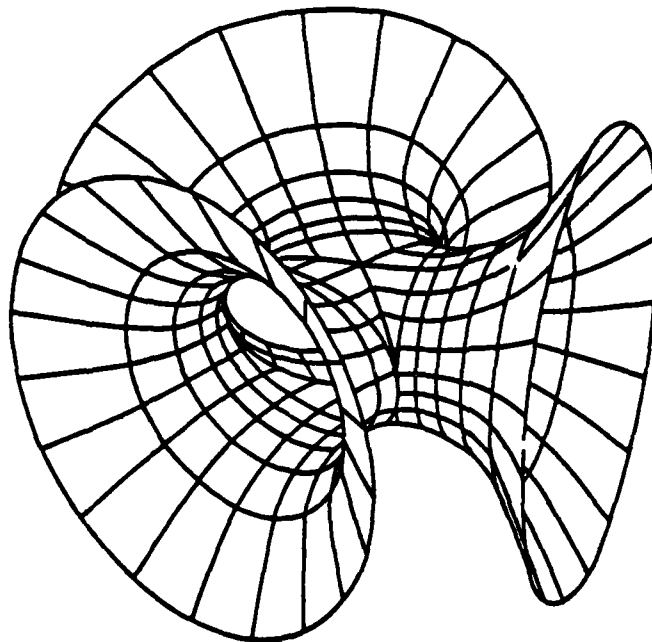


Figure 22

The effect of any of the degrees of freedom on the shape of the Jorge-Meeks surface is quite complex and is probably not easily predictable to a designer without access to either a symbolic algebra program or a nearby mathematician specializing in complex analysis.

6. SUMMARY OF RESULTS

The motivation for this paper was the use of geometric considerations in the development and implementation of blending algorithms. As was indicated earlier, such constraints have typically not been incorporated into the blending portions of solid modeling systems. The geometrical constraint that we have considered here is minimizing the surface area of a blending surface. This minimization of surface area was approximated by the use of a solution to the minimal surface equation (2.1) and a solution of the equation was called a minimal surface.

The simplest method of generating minimal surfaces was presented and examination of the method indicated some previously unknown degrees of freedom in the minimal surfaces. The effect of certain combinations of these degrees of freedom on the surfaces was indicated by several examples.

The results here can be incorporated into a solid modeling system in the following manner. The user specifies the solids to be blended and (if desired) the curves on the solids through which the required blend is to pass. The user then selects the type of fundamental blending surface: Enneper's, catenoid, Jorge-Meeks, Chen, etc. The choice is made according to the experience of the user in actually using the particular surfaces. The user then selects the appropriate degrees of freedom (a, f in the case of Enneper's surface) and selects values from avaluator device. The blends are then sketched using a large value of the increment so as to be able to rapidly reject any obviously inappropriate blends.

The number of degrees of freedom is considerably less than the number of degrees of freedom in general even for low degree surfaces; Salmon described 23 "species" for the general surface of degree 3 which of course has many coefficients. The situation is much worse for surfaces of degree 4 and hence our method makes the problem much more tractable by limiting the choices. The number of degrees of freedom can often be reduced even more using our method by noting that certain of the parameters represent rotation and scaling of the surface (e and f in Enneper's surface) and that not all parameters can be 0. The small number of degrees of freedom is appropriate for matching the blending surface to the values of specific points on the surfaces to be blended.

7. OPEN PROBLEMS

The analysis given here concentrates on minimal surfaces which blend together two given surfaces. No requirement has been made of tangency to the given surfaces. Incorporation of tangency information would require use of Lagrange multipliers in the analysis. It is expected that the blending surfaces will only approximate the actual tangency except in rare circumstances where the tangency is exact. The primary concern of this paper was the use of blending surfaces which minimized surface area. However, in a typical milling operation, the volume of excess material is also important (especially if the material is expensive).

Minimizing volume is generally impossible unless the blending surface actually follows the surfaces to be blended exactly, in which case use of a blending algorithm is obviously pointless. It is possible to consider volume minimization over all surfaces of a fixed degree, both with or without consideration of tangency to the surfaces to be blended. We intend to return to this subject in a future paper.

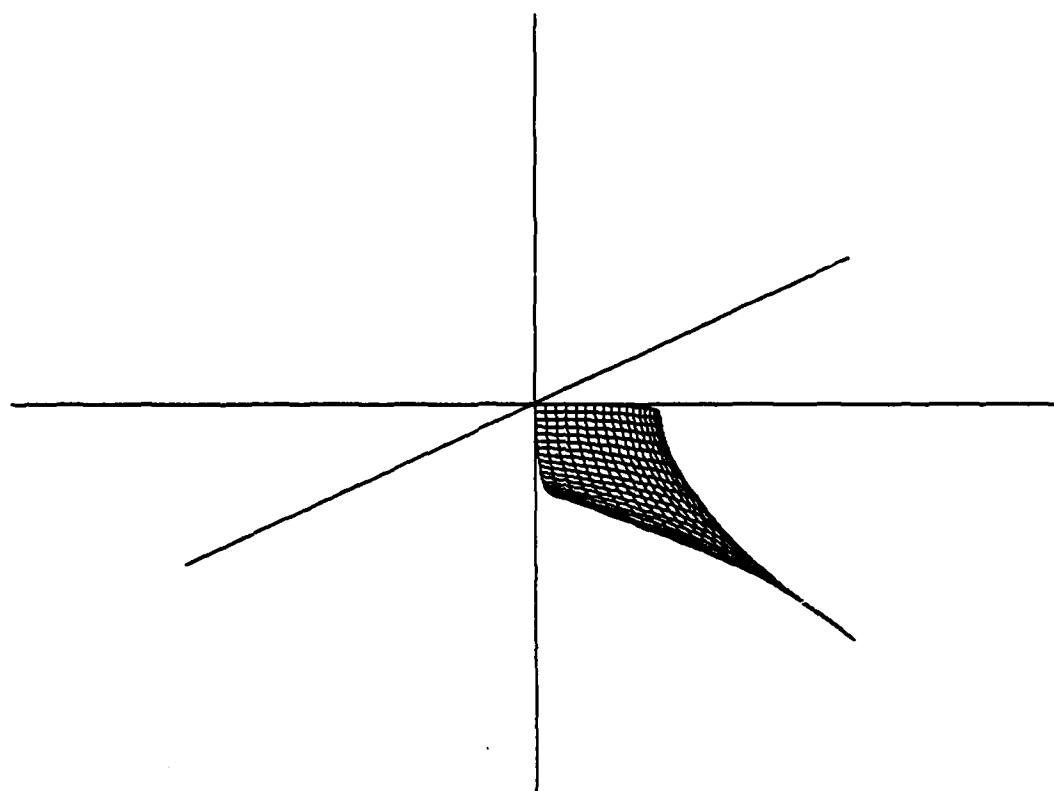
The blending surfaces described in this paper have relatively simple parametric representations yet lead to implicit equations of very large algebraic degree or to non-algebraic surfaces. It would be useful to be able to obtain a minimal surface of low algebraic degree or to verify that none are possible.

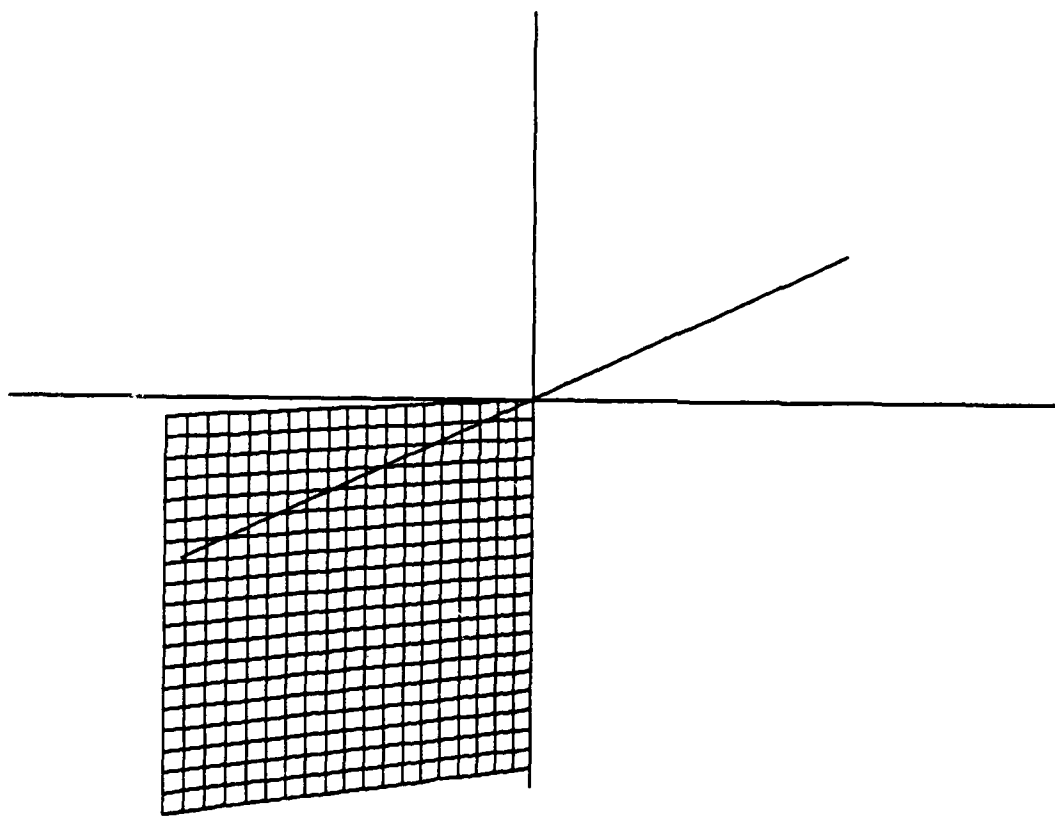
Acknowledgement

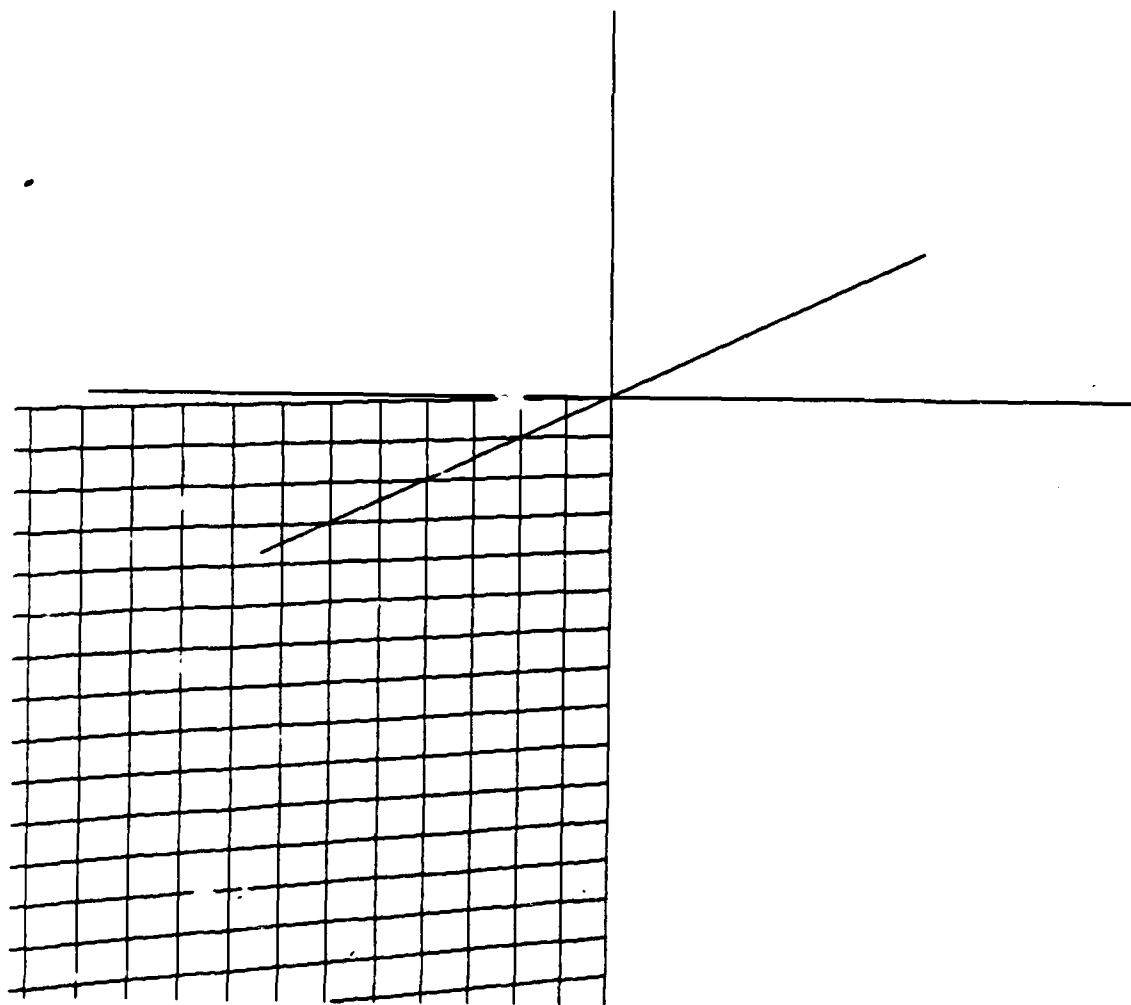
This research was partially supported by the U.S. Army Research Office under Research Grant #DAAL-83-06-G-0085

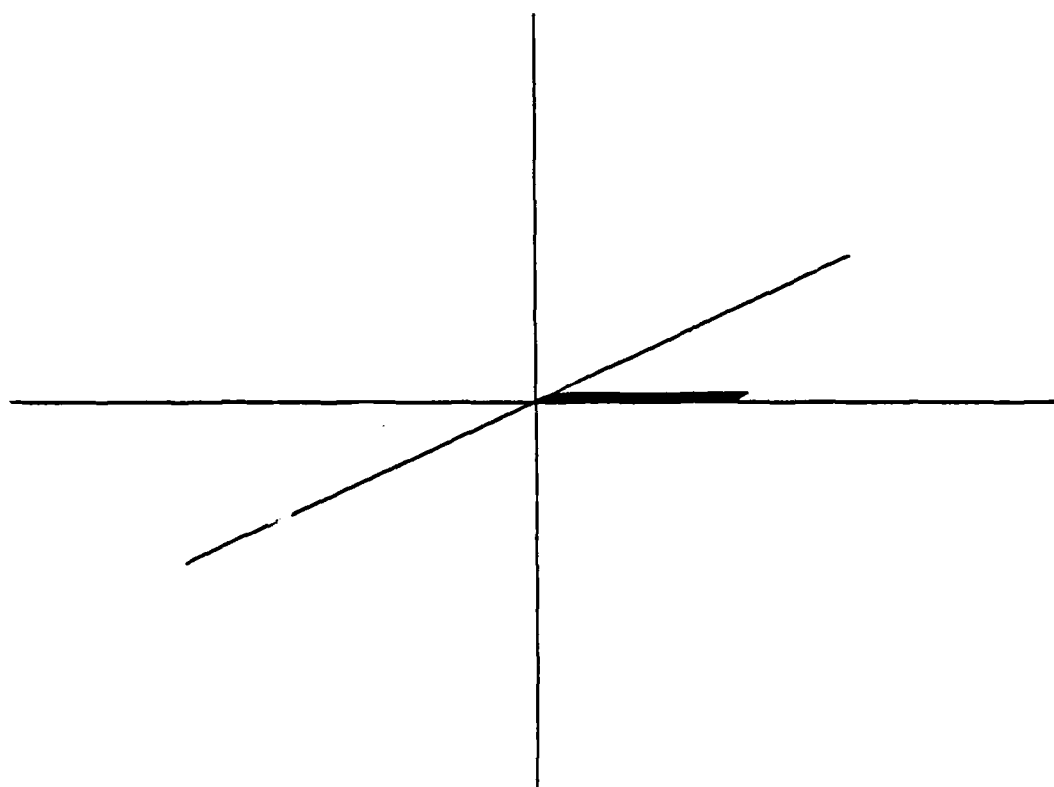
REFERENCES

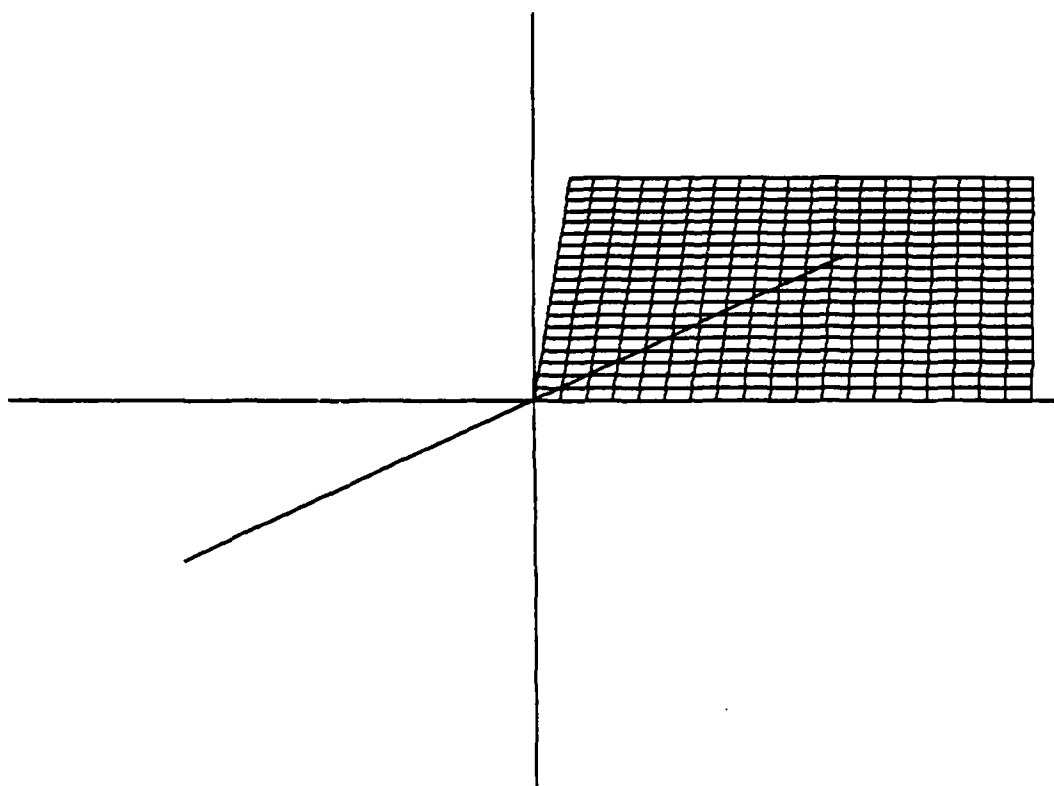
1. I. M. Gelfand and S. V. Fomin, *Calculus of Variations*, (R. A. Silverman, trans.), Prentice-Hall, Englewood Cliffs, NJ, 1963.
2. J. L. M. Barbosa and A. G. Colares, *Minimal Surfaces in R^3* , Springer-Verlag, Berlin, 1986.
3. J. L. Coolidge, *Conic Sections and Quadric Surfaces*, Oxford University Press, Oxford, 1945.
4. C. M. Hoffman and J. E. Hopcroft, *The Potential Method for Blending Surfaces and Corners*, Geometric Modeling (G. Farin, ed.) SIAM, Philadelphia, 1986.
5. L. Holmstrom, *Piecewise Quadratic Blending of Implicitly Defined Surfaces*, SIAM Conference on Applied Geometry, July 20-24, 1987, Albany, NY.
6. L. P. M. Jorge and W. H. Meeks, *The Topology of Complete Minimal Surfaces of Finite Total Gaussian Curvature*, *Topology* 22 (1983), 203-221.
7. E. Kreyzig, *Differential Geometry*, University of Toronto Press, Toronto, 1959.
8. R. Leach, *Geometric Considerations in Blending Surfaces*, to appear.
9. U. Massari and M. Miranda, *Minimal Surfaces of Codimension One*, North-Holland, Amsterdam, 1984.
10. A. Middleditch and K. Sears, *Blend Surfaces for set Theoretic Volume Modelling Systems*, *SIGGRAPH Comp. Graphics*, 19, 1985, 161-170.
11. M. Mummy, *Automated Constant-Radius Blending in a Boundary-representation Solid Modeller*, SIAM Conference on Applied Geometry, July 20-24, 1987, Albany, NY.
12. A. Rockwood and J. Owen, *Blending Surfaces in Solid Geometric Modeling*, SIAM Conference on Geometric Modeling and Robotics, 1985, Albany, NY.
13. G. Salmon, *A Treatise on the Analytic Geometry of Three Dimensions*, Longmans Green and Co., New York, 1915.
14. D. M. Y. Sommerville, *Analytical Geometry of Three Dimensions*, Cambridge University Press, Cambridge, 1934.

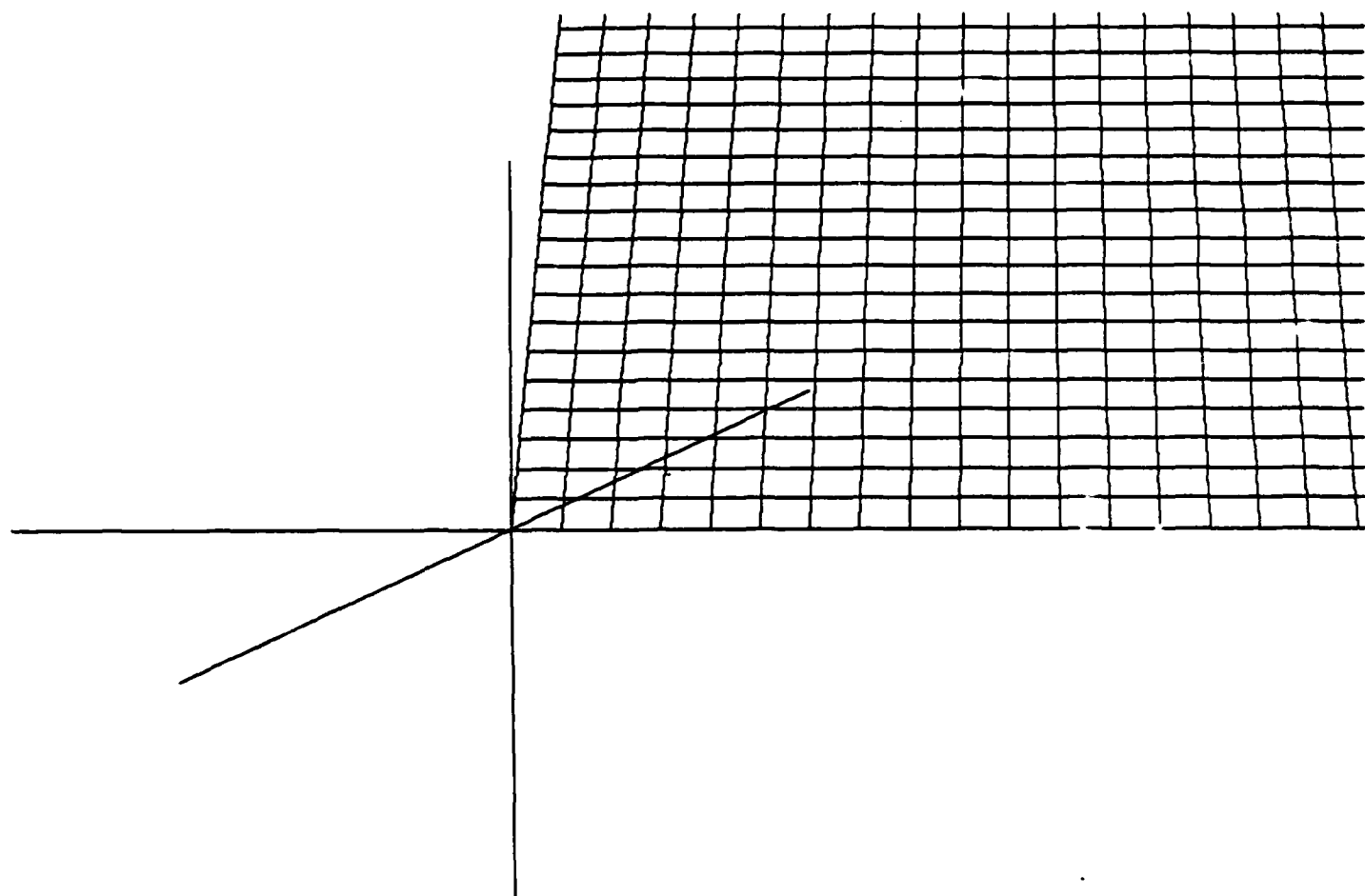


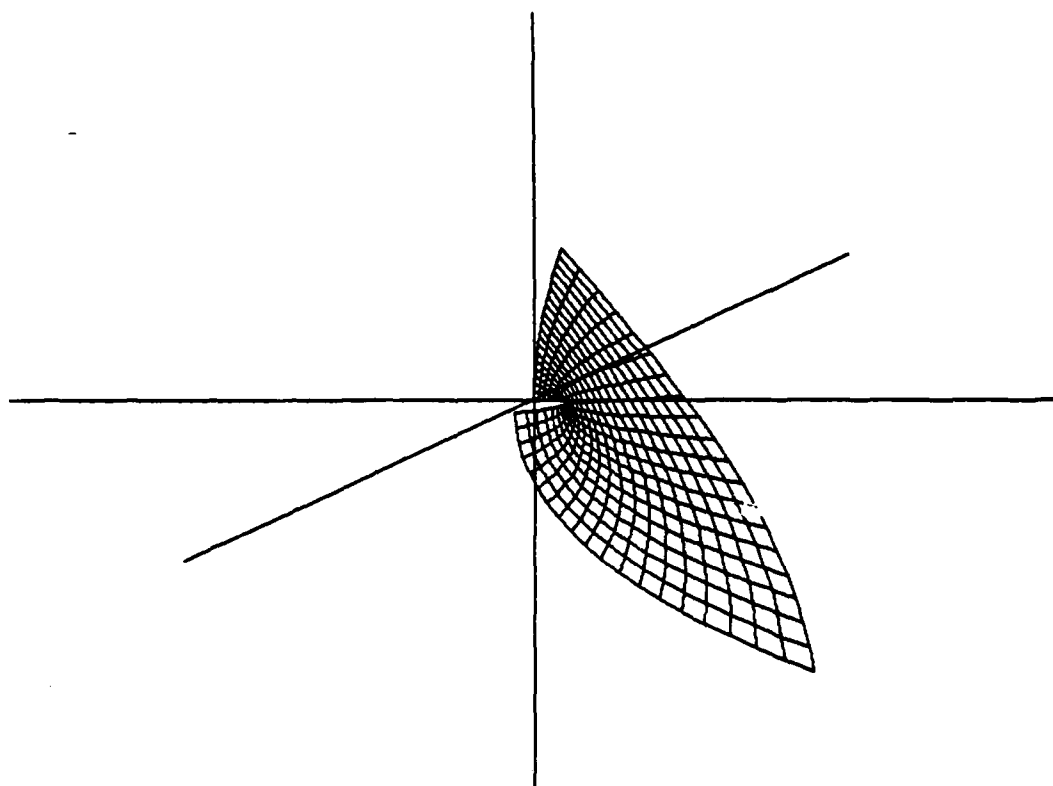


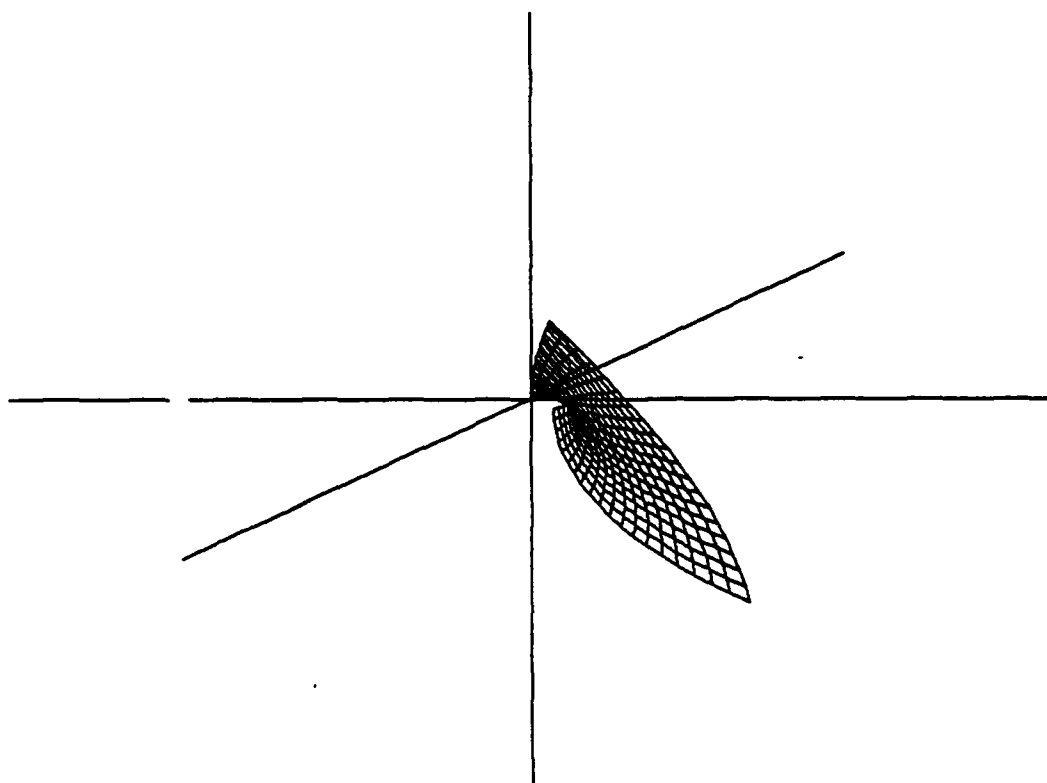


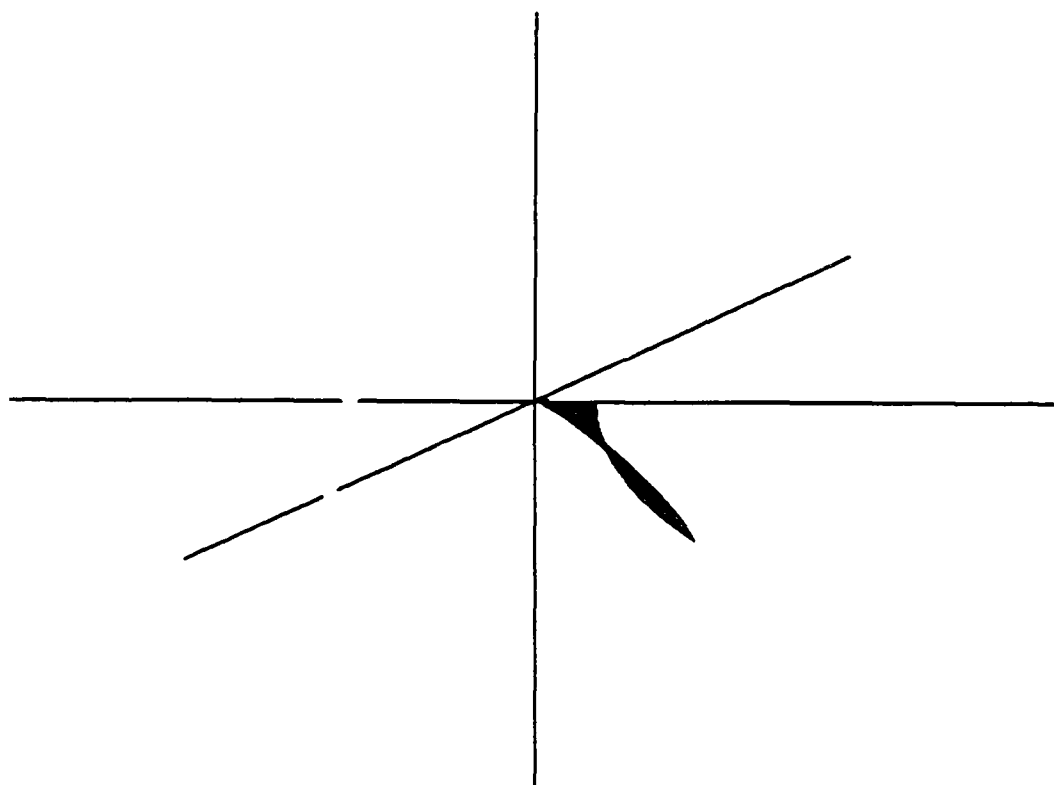


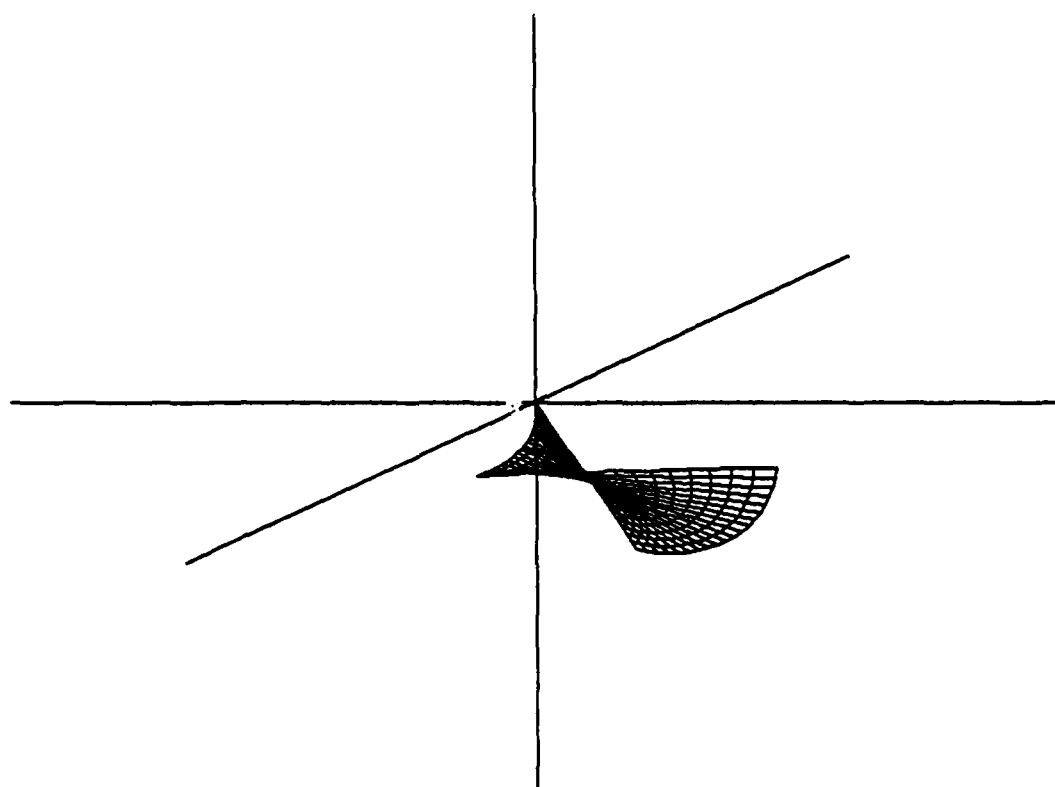


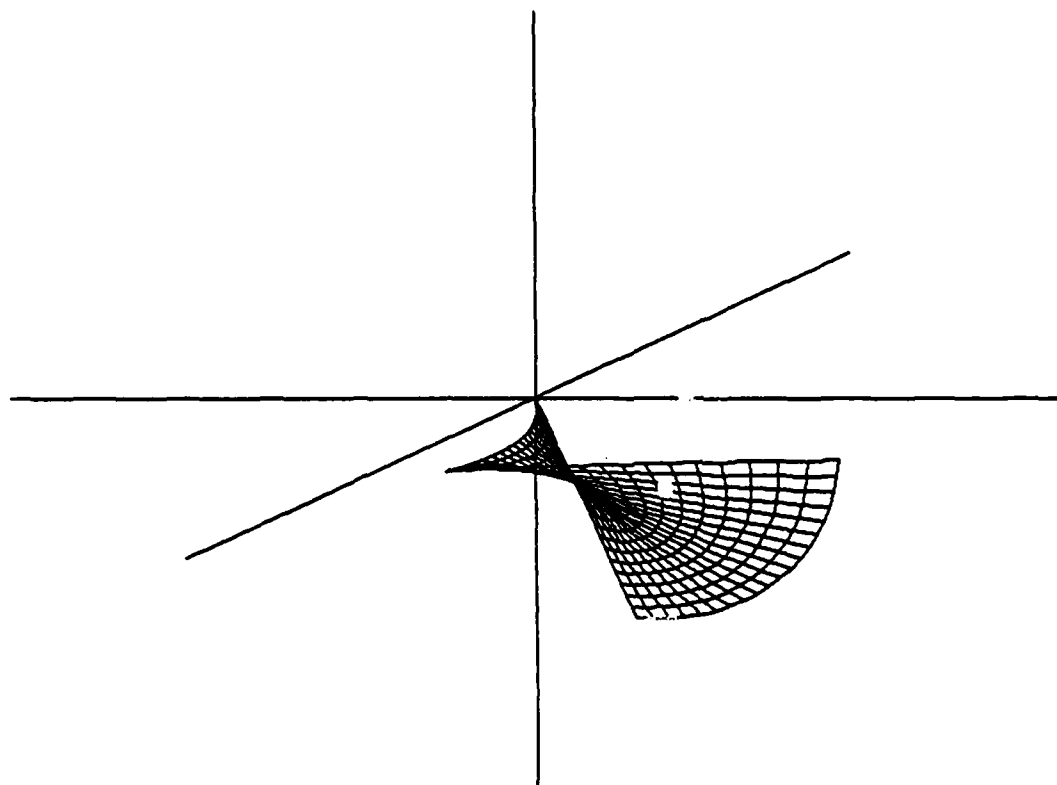


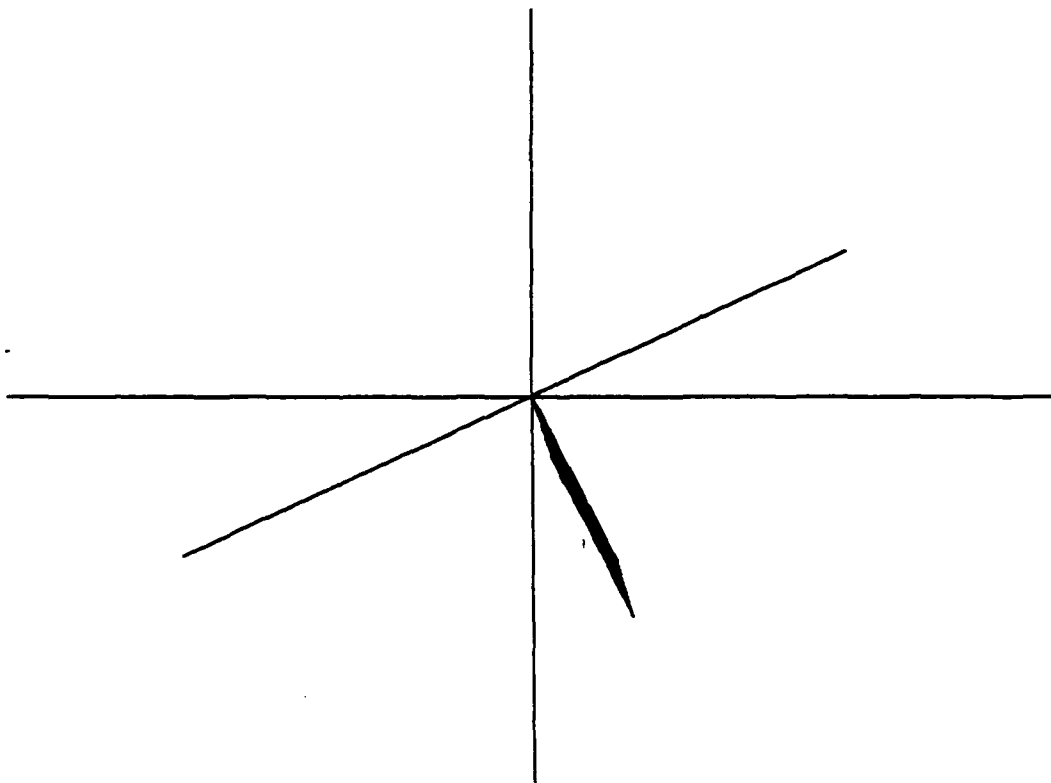


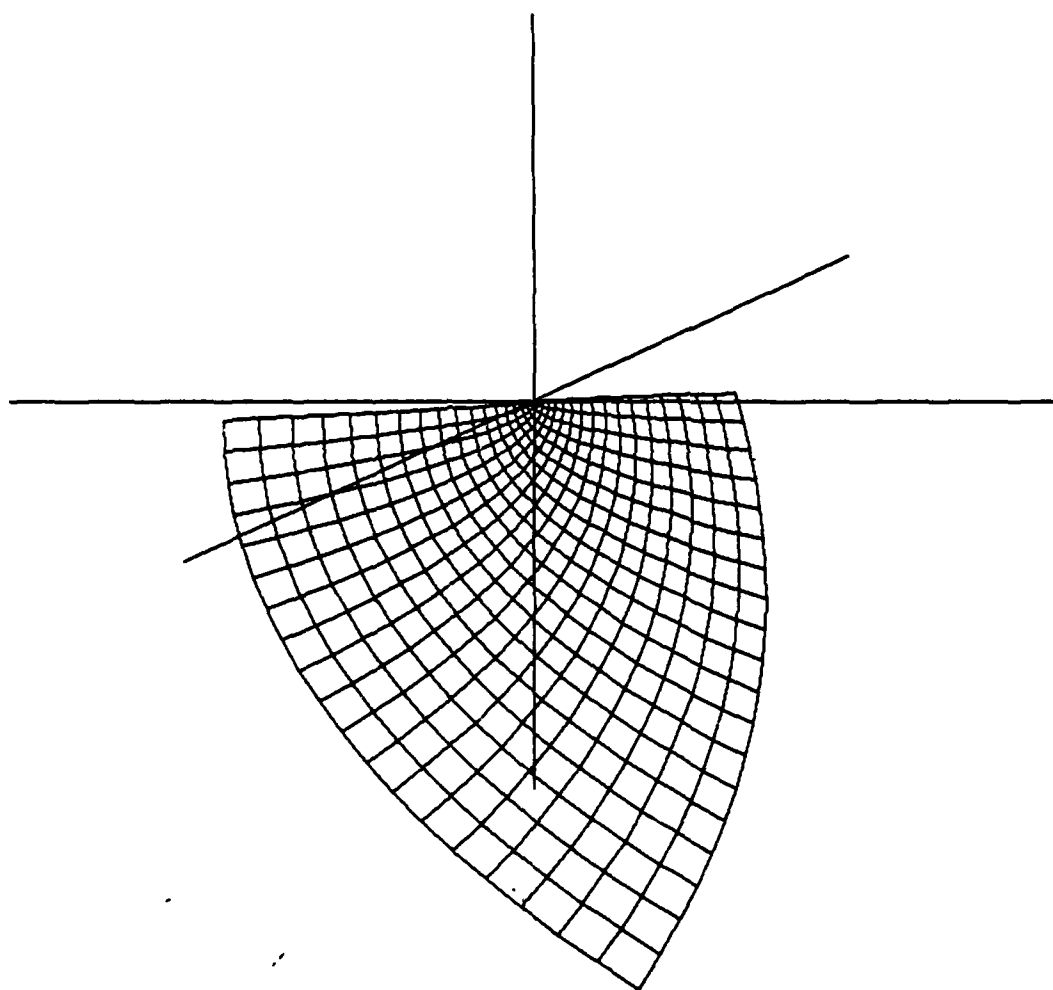


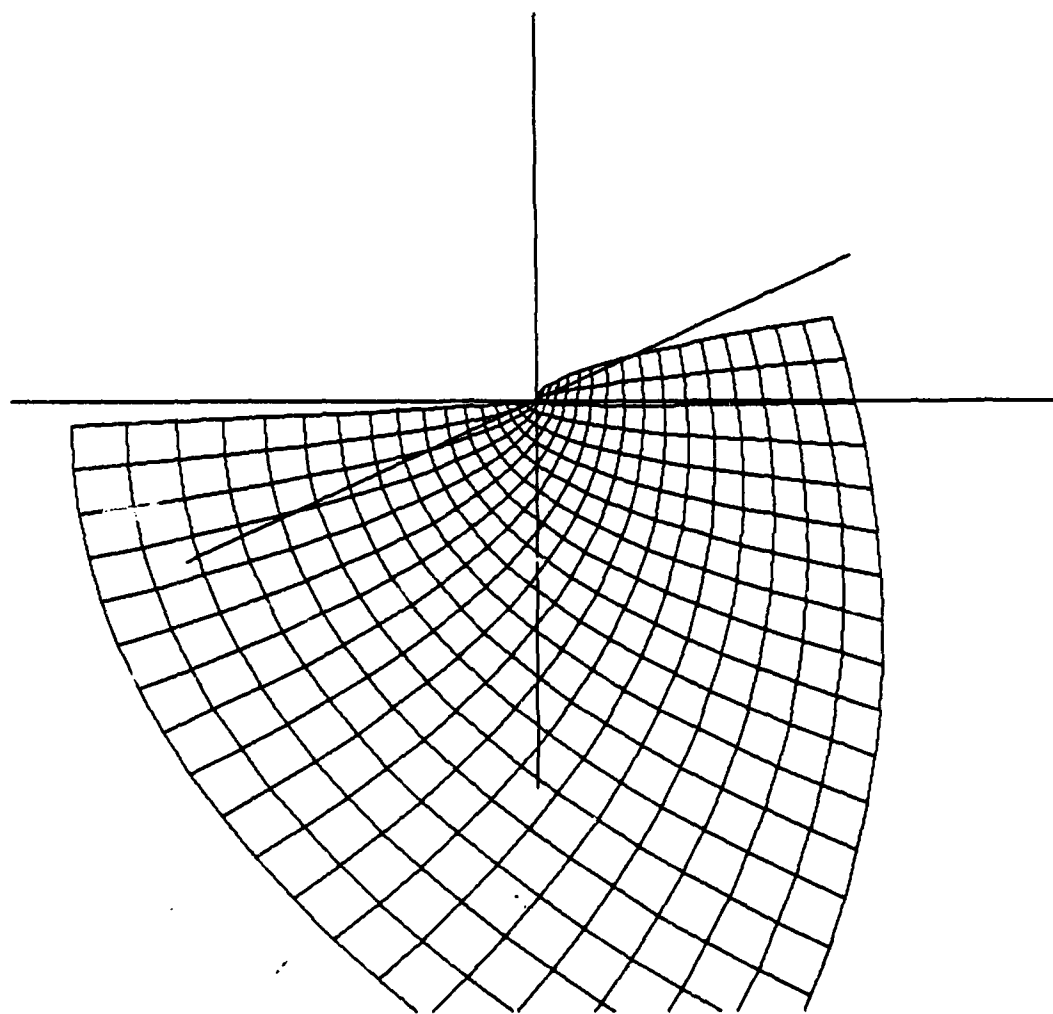


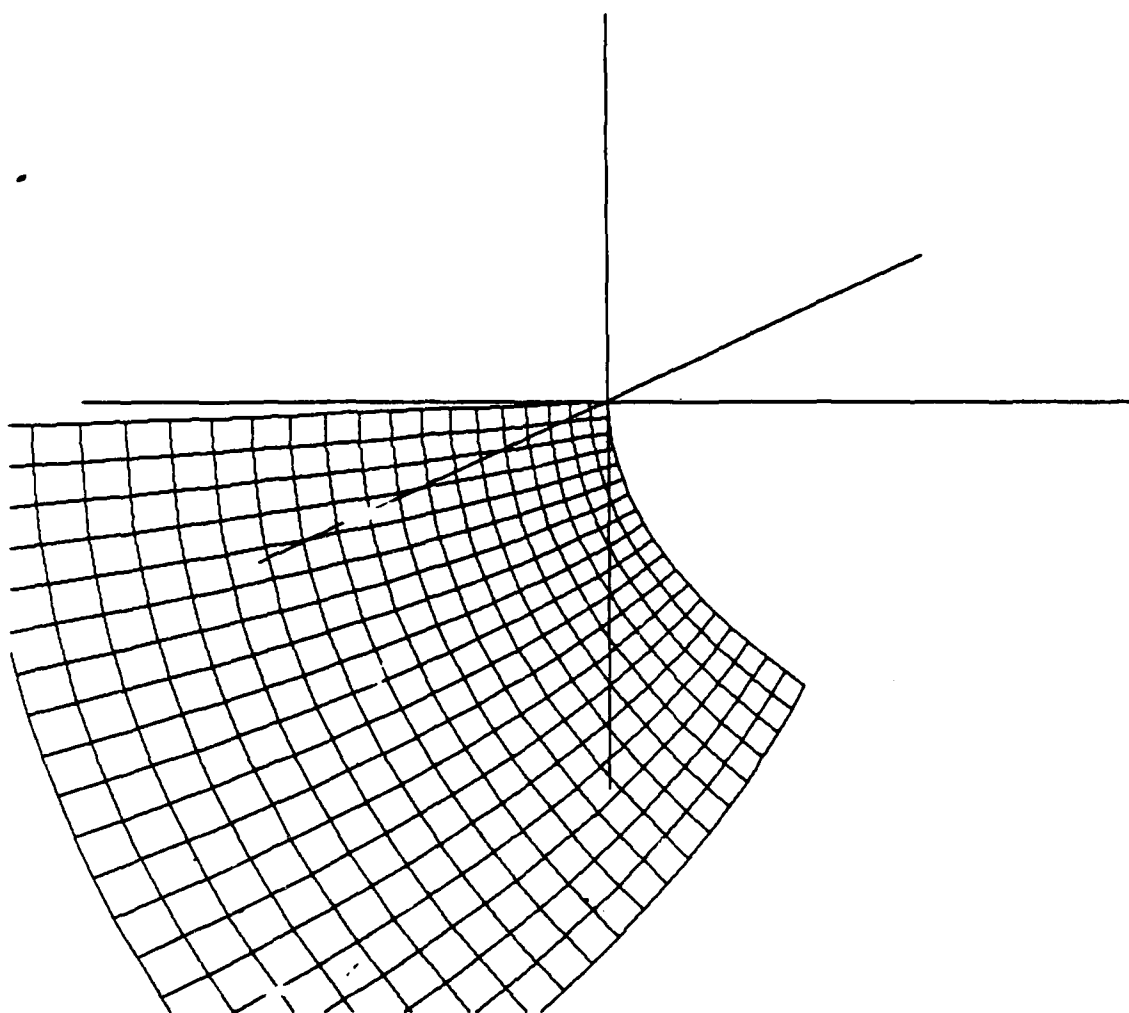


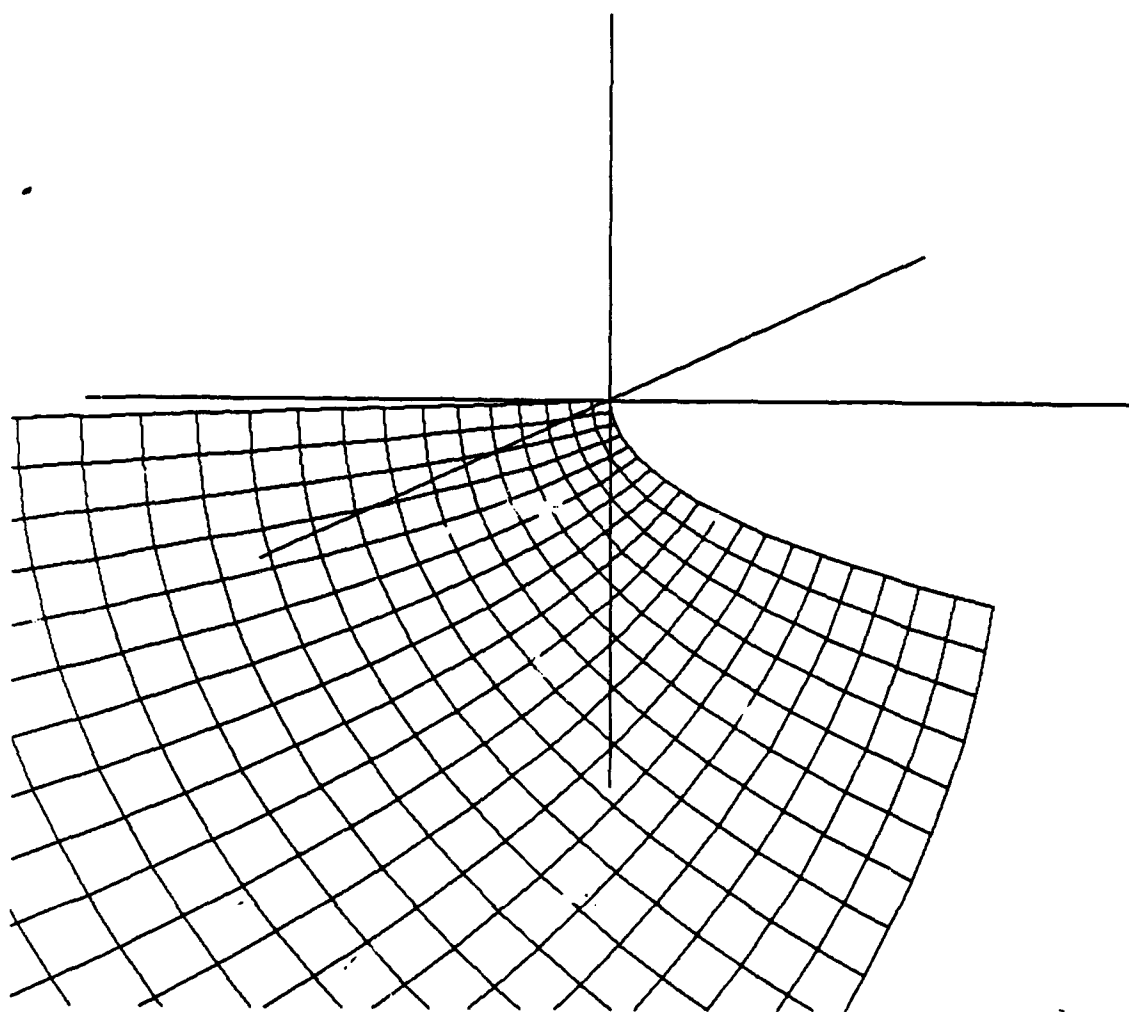


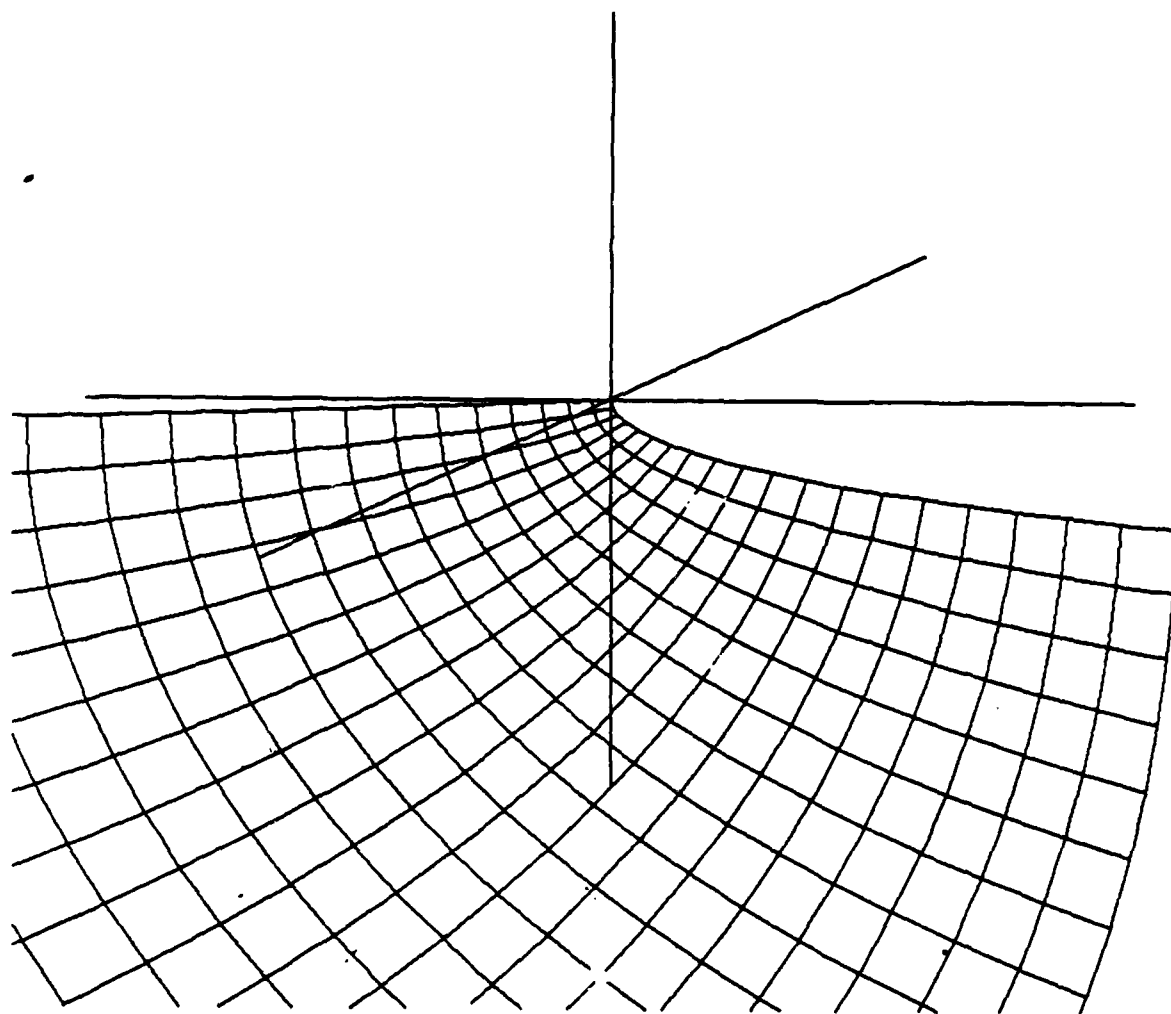


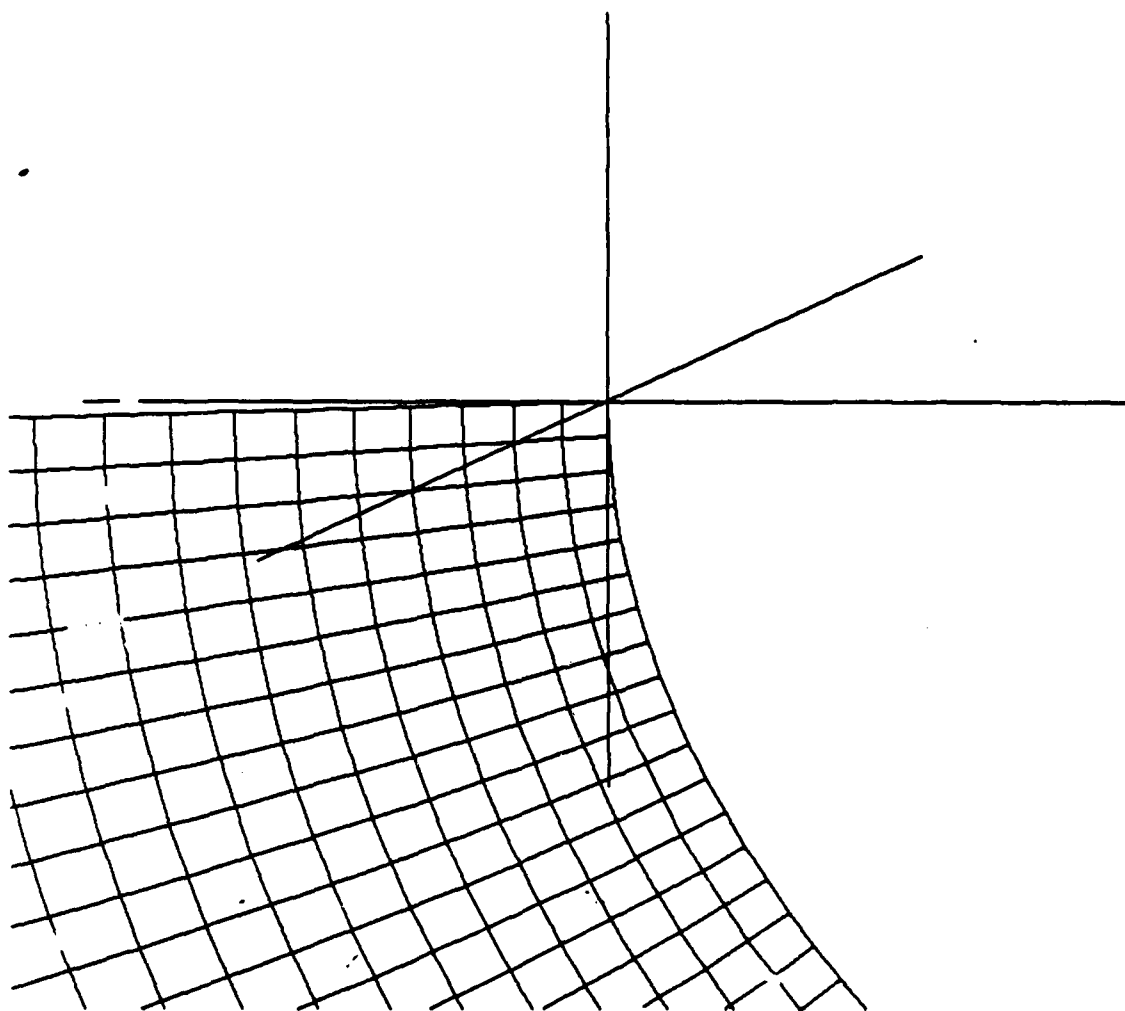


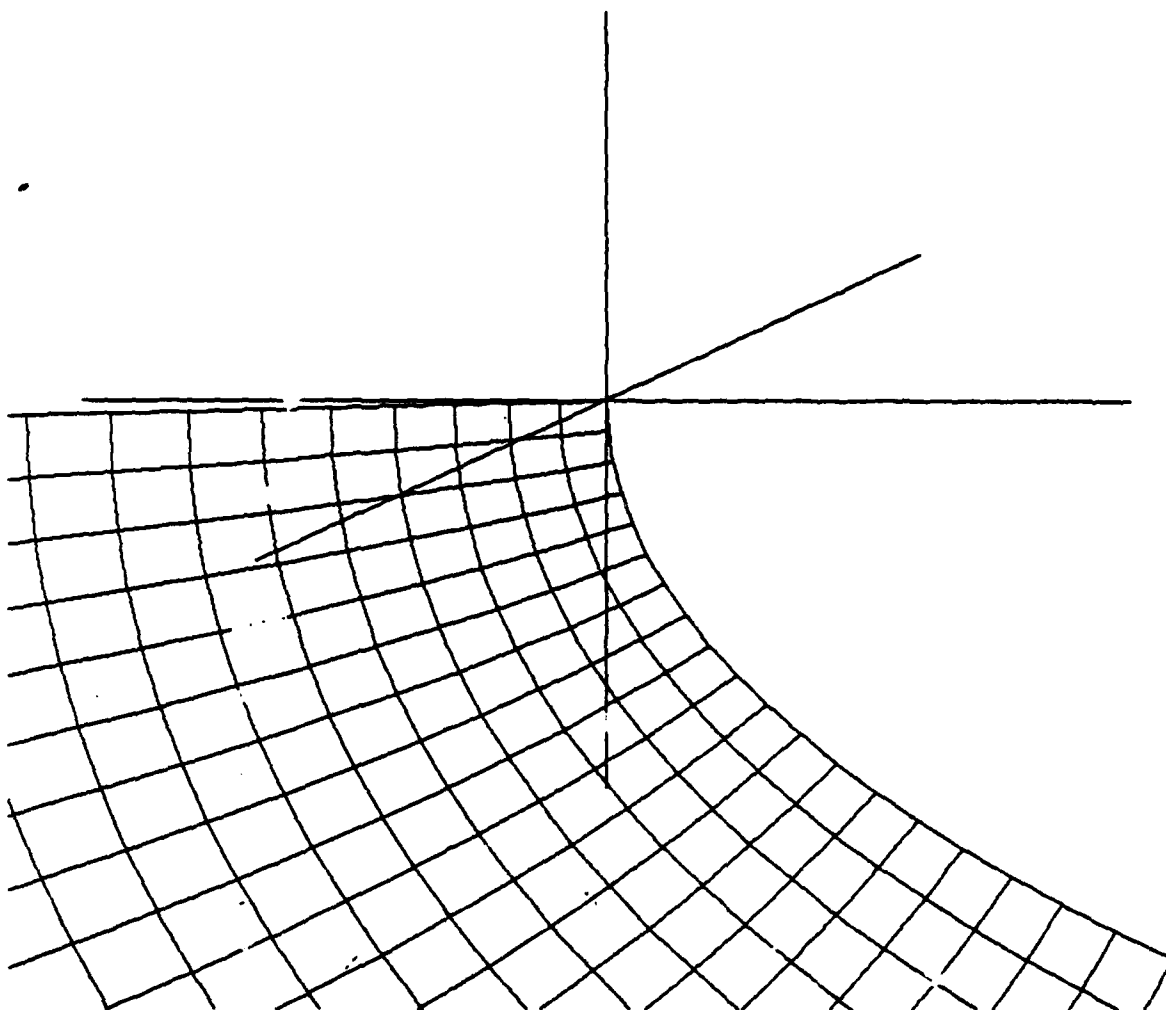


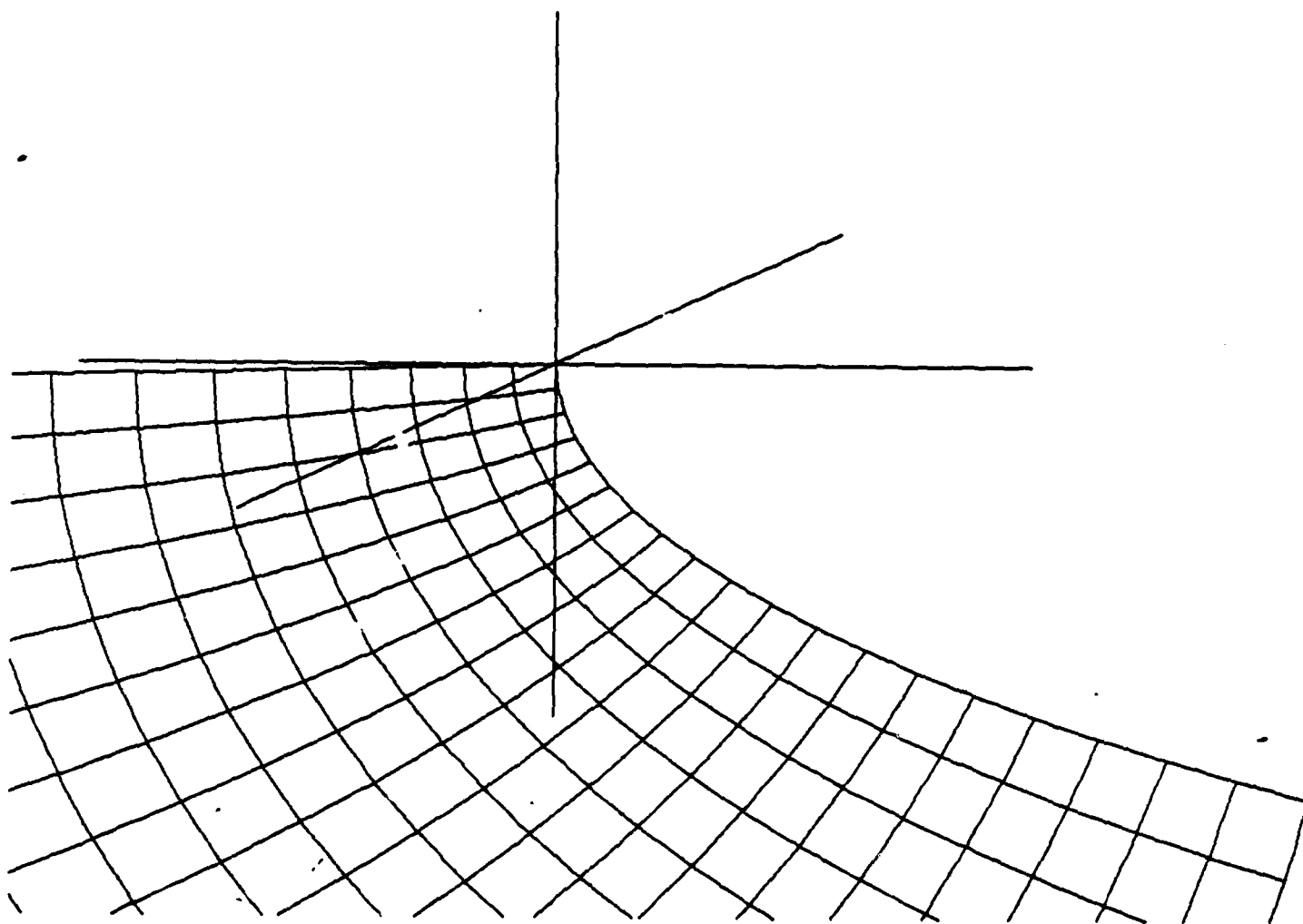


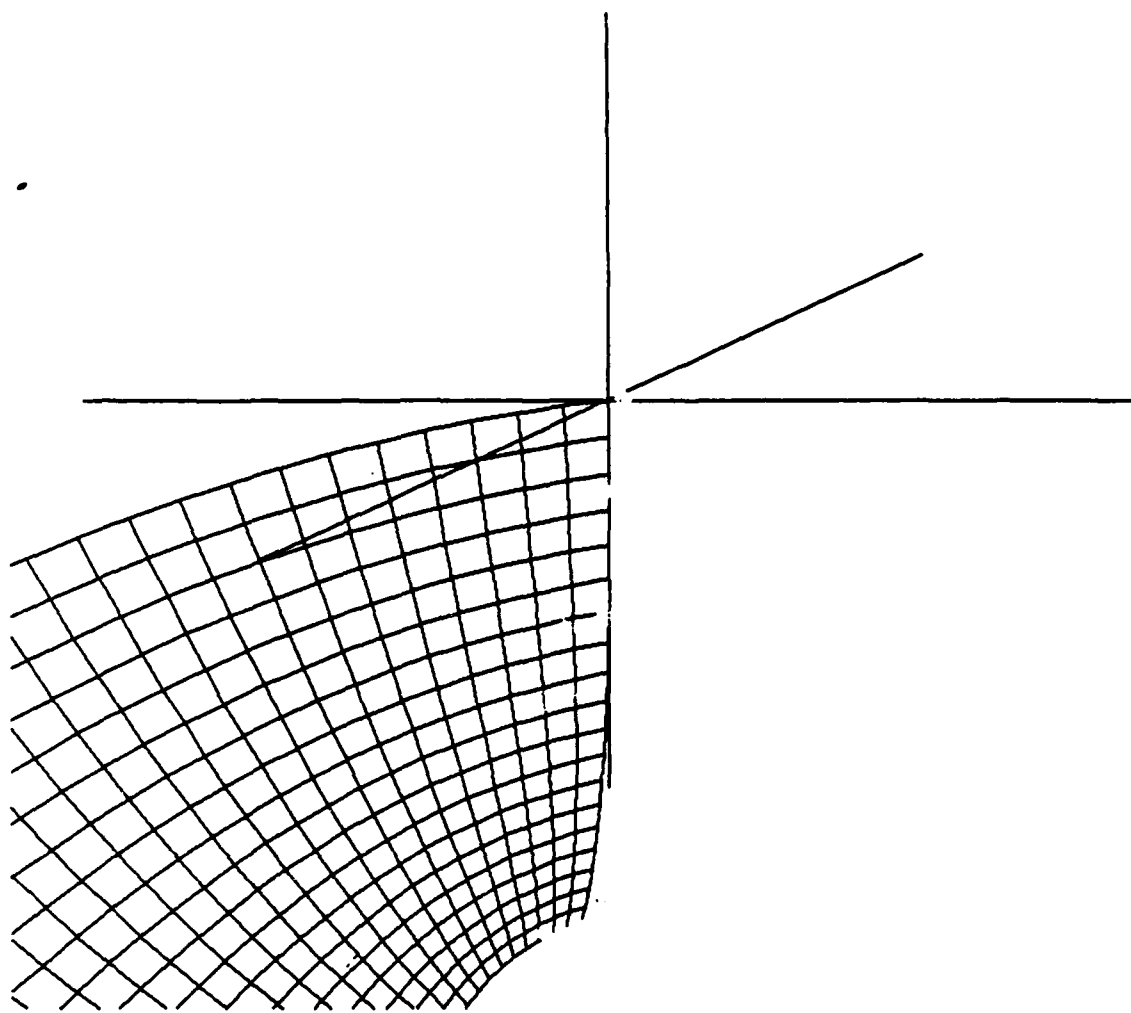


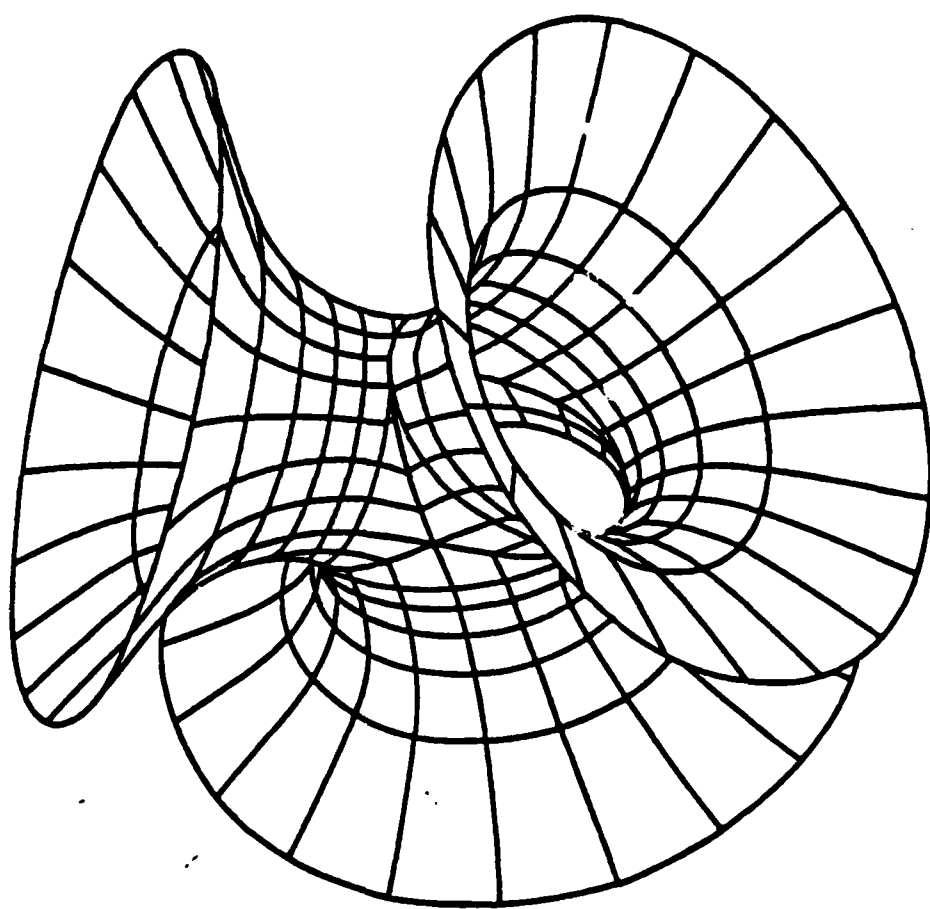












EXPERIENCES TEACHING CONCURRENCY IN ADA

Ronald J. Leach
Department of Systems and
Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

ABSTRACT

Many students have great difficulty understanding concurrent programming at anything but the most superficial level. In this paper, we describe some experience teaching concurrent programming in Ada and give some suggestions for implementing the ideas discussed here.

1. INTRODUCTION:

The concept of concurrent programming is one of the most difficult ones for students to understand. In general, upper level students have a good understanding of structured design, analysis of algorithms, and a rudimentary knowledge of software engineering. Such students generally are proficient in several different programming languages that are used for sequential programs. In this paper, we describe some experiences when having students learn about concurrent programming by writing programs in Ada. While the environment we describe is specific to a particular course at Howard University, many of the experiences encountered can be carried over to other environments.

2. THE ENVIRONMENT:

The course in question was an advanced course in programming

languages taught to seniors and graduate students at Howard. The undergraduate students were majoring in Computer Systems Engineering and the graduate students were majoring in Computer Science; all students were majoring in the School of Engineering. All students were proficient in the languages FORTRAN, Pascal, and C before beginning the study of Ada. In addition, many of them were exposed to a variety of other languages including LISP, PROLOG, or PL/1.

The topic of concurrent programming was introduced by means of some standard examples. The notion of Petri net was presented and used as a framework for the discussion of concurrency in tasking. No other more formal model of concurrency was used. This method of presentation is in agreement with the philosophy in [1] and [2].

The Ada environment at Howard is not conducive to the development of large Ada programs. Most of the Ada assignments are done on a Digital Equipment Corporation VAX 1/780 with 4MB of main memory which runs under VMS. The "compiler" is the classic NYU AdaEd. Because of the size of the working set needed for compilation under this compiler, students were encouraged to submit jobs for batch processing. Interactive compilations and executions were limited to one terminal at a time, since working sets of 2MB were allocated to Ada processes. In addition, students learned Ada syntax by writing small Ada programs using the limited Janus Ada on personal computers. There were 21 students in the class. A much larger class would have been unmanageable for this project. Note however that the assignment of different projects eliminated the propagation of correct solutions since each of the students had a tasking

synchronization problem that was at least superficially different from that of the other students.

3. THE PROJECTS:

The students were each assigned a project for which they were required to write Ada programs which involved at least two tasks. In general, the tasks embodied some simple idea that the students were very familiar with at least in the case of sequential programs. Thus the difficulty was in understanding the concurrency and not in the computation performed by the individual task.

As an example, one student was asked to write a program with two tasks - sort an array of integers and then search for a key using a binary search. The student was allowed to use any sorting algorithm. Thus the student did not have difficulty implementing the individual algorithms for the tasks. The troublesome part was the implementation of the synchronization or communication of the tasks.

Projects involving similar, but not identical problems in synchronization were given to other students. Some examples of these assignments are:

1. Write a program which uses two tasks to solve quadratic equations using the quadratic formula. Each task must perform at least three arithmetic operations.

2. Write a program to read an integer n and to have two tasks. The tasks are to compute some simple function $f(n)$ and to find all primes less than $f(n)$.

3. Write a program to simulate the donning of socks and shoes. Putting on socks and putting on shoes are to be separate tasks.

4. Write a program to read an array A of integers and to have two tasks. The tasks are to sort the array A in increasing order passing this sorted array to B and to sort the array B in decreasing order.

Clearly the major difficulty for the students was the synchronization of tasks. Students were required to run their programs four times with the same input. Most of the errors in the programs due to subtle assumptions about tasking made by the programmer became apparent after four runs.

Student's observations on this point were interesting. In spite of several lectures on timing and synchronization of tasks, lengthy discussions on the nature of an Ada "logical processor", and numerous classroom examples, students did not believe that programs could give different results or bomb when given the same input. The sudden shock when their program showed this behavior put the point across better than any lecture could. Many of the students indicated that they had seen this kind of error at some time during program development. Two of the students were so shocked by the different behaviour of the sample runs that they turned in their projects with signatures of witnesses that their programs ran successfully, at least once.

4. CONCLUSIONS AND SUGGESTIONS FOR IMPLEMENTATION

The students who had been through this assignment seemed to have a better understanding of concurrent tasking in Ada than did students in previous semesters. The assignment of programs involving several (perhaps trivial) tasks which needed to be synchronized or communicated. This can be implemented in several ways.

- 1) Assign different projects to students (or to small groups of students) requiring them to have at least three or four runs of their program.

- 2) Assign the same project to different students (or groups) and have them compare sample runs on the same data. This should point out difficulties in tasking.

- 3) Write a program yourself to do one of the assignments given above. Don't think carefully about all possible orders of execution of the tasks. Your program is likely to have different outputs depending on the actual physical implementation of the tasks.

REFERENCES

1. Cherry, G., "Parallel Programming in ANSI Standard Ada", Reston, Reston, Va., 1984
2. Gehani, N. "Ada : Concurrent Programming", Prentice-Hall, Englewood Cliffs, N.J., 1984

EVALUATING THE PERFORMANCE OF A USER INTERFACE

RONALD J. LEACH

Department of Systems and Computer Science, School of Engineering,
Howard University, Washington, D.C. 20059

Abstract—A considerable amount of research has been done in the area of human-computer interaction. High quality human-computer interfaces are especially important when a program needs to perform in nearly real-time. In this paper, a particular human-computer interface is analyzed. The observations made are abstracted and generalized to problems in interfaces for process control. Specific benchmarks for evaluation of interfaces in a real-time or near real-time setting are developed. We show that these benchmarks are more useful than general performance measures of computer systems for estimating computer performance. We develop general guidelines for designing interfaces which satisfy severe time constraints and determine to what extent they follow generally accepted principles for interface design.

1. INTRODUCTION

A frequent use of computers is in the control of processes. In many situations, processes that were controlled by a group of individuals are now controlled by one or more computers with human operators performing a monitoring function. The operator takes control only for short periods of time for testing or when an emergency occurs. The human-machine interface is especially important in such a situation. Poorly designed systems produce operator boredom and often cause operator errors. At critical times, the response time of a poorly designed system may be too slow for effective process control.

There is a large and growing body of research on human-computer interfaces, with some design principles beginning to emerge. Much of the research in this area is based either on laboratory experiments or is anecdotal and based on observations of adherence to general principles. See the article by Foley, Wallace and Chan [1] for an excellent survey of research on the psychology of human-computer interaction with emphasis on the area of computer graphics.

Often the best user interfaces require extensive use of computer graphics displays. Using graphics in the interface increases the load on the computer system and requires an expenditure for hardware that ranges from very small (personal computers) to quite large (high performance color graphics workstations). Thus the design of a user interface must take into account several factors such as availability and cost of graphics hardware, demand on the computing system, and any requirement for real-time performance.

There is also a large and growing body of research measuring the performance of computer hardware and software. Results in this area are often results in the areas of computer architecture, analysis of algorithms, simulations, reliability models, or some combination of these areas. See [2], [3], [4], and [5] for typical results in these areas.

In Section 2, we will briefly discuss the evolution of several generations of a particular human-computer interface. This particular interface was studied by the author at the Goddard Space Flight Center of NASA while the author was on a NASA/ASEE Faculty Fel-

lowship. The results are extended and generalized greatly in the rest of the paper. Particular emphasis is given to the development of benchmarks (Section 3) and guidelines for evaluating and predicting performance (Section 4). In general, we show how commonly accepted principles of design of human-computer interfaces can be combined with specific design requirements without significantly degrading performance in many instances.

2. A TYPICAL EXAMPLE

A control room at the Goddard Space Flight Center of NASA is responsible for the control and operation of many spacecraft. A typical spacecraft sends back information to ground telemetry stations on a regular basis; the amount and type of information varies when certain experiments or ground tests are performed. This telemetry information is then relayed to a ground computer system which consists of various processors dedicated to recording telemetry data sent back by a spacecraft, executing applications programs and controlling communications interfaces to various other devices. Generally, these ground computers have a moderately heavy steady-state computing load, which is well understood at the time that a mission is planned. There is considerable hardware redundancy; an applications processor is kept in reserve in case of failure. Because of the design requirements, any graphics must be done locally, with little or no load on the applications computers. Information about the control of the spacecraft must be available to the control system within two seconds, with real-time response desirable.

Originally, a control center would have had many monochrome monitors displaying alphanumeric information. Redundancy was provided by having more than one person look at the same information. These displays created little demand on system resources. However, training of operators was slow and there was a relatively high rate of error and fatigue, even for experienced operators. An operator must monitor many existing experiments and must initiate others in order to assure that the spacecraft is functioning properly.

A large amount of human factors research has been applied to the design of control rooms. In particular,

the use of graphics displays and color has had a pronounced effect on operator training and efficiency. The current trend in control rooms is to provide more information to fewer operator/analysts by using color computer graphics displays on CRTs. See [6] and [7] for more information about the evolution and design of these displays.

Use of graphics displays in control room setting falls into three main categories: monitoring of processes, testing and control, and display of moving objects. Monitoring of processes includes plotting of two dimensional graphs. Typical graphs are plots of reserve levels of batteries or fuel consumption over time. Monitoring of processes also includes showing configurations of ground computer networks and availability of various computers. This type of activity does not require state of the art graphics equipment. The principal need is for rapid screen updating and display of stored graphical data. Current plans are to use IBM PC ATs with special graphics cards and storage devices for these displays [8, 9].

Testing and control involves operator/analysts deciding which tests to perform on a satellite or spacecraft. Control rooms for current flights use a touch screen on a CRT display of a "command panel" which presents many of the options of tests to be run. These command panels have evolved from alphanumeric displays in which an operator typed in the test that was to be performed. There is some consideration of saving expensive screen "real estate" by using functional tablets which have some of their commands preset.

The major issues here are interface technology, ease of learning, and ease of use once the command system has been mastered. Touch screen technology provides reasonably fast sensing at moderate cost. (256 by 256 touch points and 50 points per second conversion rate is a typical example [10].) Operator satisfaction with the ease of learning and use of this system is high. Anecdotal evidence that a touch screen is preferable to a light pen agrees with the findings of [11] and [12]. We note that there is no need for expensive graphics processors with huge display list memories in the case of testing and control. The primary need is for input devices and relatively inexpensive graphics workstations.

The most exacting use of computer graphics in a control room at NASA is in the display of moving objects such as the space shuttle. The simplest way to display the shuttle involves using a wire frame model with no removal of hidden lines. Adding the antennas, cargo bay doors, and robot arms complicates the picture so that hidden lines must be removed in order to make the picture useful. Allowing for three dimensional rotation means that each vertex in the model of the shuttle will require the multiplication of a vector by a matrix. For a reasonably useful wire frame drawing, this will use up approximately one half of the cpu time of a minicomputer such as a VAX 11/780, which is completely unsatisfactory. Using color graphics representation of solid objects with hidden surface removal and some shading greatly increases the computing load.

The only reasonable solution is to use a powerful graphics workstation with hardware matrix operations and many of the necessary algorithms in hardware. See [2], [3], and [13] for typical estimates of computer load caused by three dimensional rotation.

3. EVALUATION OF AN INTERFACE: BENCHMARKS

In Section 2 of this paper, we discussed a control center as an example of a user interface and the environment in which it is used. The evolution of user interfaces in this setting was discussed. We now turn our attention to the evaluation of the performance of general interfaces in a real time setting. In this section, we discuss the hardware and software requirements of interfaces, with emphasis on benchmark programs. Our primary emphasis is on evaluation of interfaces which control large numbers of relatively static displays for monitoring of processes or for testing and control. Interfaces for display of the motion of complex objects will not be considered in this paper.

The first question to be considered is how we will be able to store the displays. We assume that the logical organization of the displays is hierarchical, with relatively few levels and a fairly large number of children possible from each parent node. Results in [12], [11], and [6] suggest that this is the most effective design for the menu selection.

We note that there are only a few possibilities for the storage of data for any display: as collections of pixels in either compressed or uncompressed form or else they can be stored as collections of instructions to a display processor. The various displays can be shown as entire screens, portions of a screen in a window, or on several screens. Selections can be made from menus that range from fixed and permanent as in the case of hard-coded touch tablets to relatively fixed such as in touch panels on CRTs and finally to menus that are rarely visible as in pop-up or pull-down menu systems.

Only a few colors are needed for a color display of monitoring and testing of processes. For example, research at NASA indicates that 5 or 6 colors suffice for control room use [6]. Thus the major factors in using bit mapped displays for monitoring and testing are screen resolution, size of primary and secondary memory, speed of access of primary or secondary memory, speed of screen updating, and of course the nature of the displays and the way they are stored. Much of this information can be obtained from manufacturer's specifications, at least to a first approximation. However, the performance of an interface also depends on the speed of various input devices and the speed and ease with which a human user can interact. Thus the evaluation of the speed of an interface in a control system which needs to approximate real time response can best be estimated by a set of benchmark programs.

To aid in the determination of actual running times of the various portions of a user interface, we will use a notation similar to but not identical to that of [4]:

- c = time to draw a circle,
- $p(j)$ = time to draw a polygon with j sides ($j > 1$),
- t = time to draw a character,

any of the above symbols followed by an f denotes the time to fill the object (to fill a character means to fill the smallest rectangle defining the character).

Finally, we denote the time to write a segment S as

$$\text{TIME}(S) = \text{sum of all } (c + p + t + c f + p f + t f) \\ \text{for all objects in } S.$$

In [4], formulas were presented for measuring the total time to display all possible displays in a large scale system for monitoring and process control. In that paper, we were interested in the way that these times are modified when a display is stored as a segment in a structured display file, has to be searched for in a tree, is in secondary storage, or needs to be compiled. Our goal here is different, since we are interested in the interface rather than the speed of rewriting displays. Thus we will ignore the factors DFILE (used for segments in a structured display file), SECFILE (for secondary storage), or COMPILE (for compiling a segment). We will need to use the factors LEVEL and

SEARCH which are related to the actual selection of a display. We will use the formula

$$\text{TIME} = (\text{LEVEL} + 1) * \text{SEARCH} * \text{TIME}(S)$$

to represent the time to write a display to the screen. Here SEARCH represents the time for searching a menu and either selecting the desired object for viewing or determining that the object is not present. LEVEL will serve as a counter for the length of the path from the root of the tree to the menu for the desired display. By convention, the length of a path from the root to itself is 0.

These benchmark are in the form of pseudocode programs which will be grouped into sets of three as follows. The first three programs show the speed of replacing a screen of text by another screen of text. These programs will use a procedure TEXTSCREEN which will write 20 lines of 40 characters each. The command SYSTEM(TIME) will provide the elapsed time since the last call to SYSTEM(TIME). The next set of three programs will be obtained from the first set by changing graphics instead of text. The last two sets will duplicate the first two sets in a window environment.

```
PROGRAM REPLACE_TEXT;
(* TIME FOR UPDATING OF TEXT PAGE WITH NO INPUT *)
BEGIN
  TEXTSCREEN;
  OLDTIME := SYSTEM(TIME);
  FOR I := 1 TO 100 DO
    BEGIN
      CLEAR;
      TEXTSCREEN;
    END;
  NEWTIME := SYSTEM(TIME);
  TEXT_TIME := ( NEWTIME - OLDTIME ) / 100 ;
  RETURN(TEXT_TIME) ;
END.
```

```
PROGRAM SEARCH_REPLACE_TEXT ;
(* TIME FOR UPDATING OF TEXT PAGE USING SELECTING DEVICE *)
BEGIN
  TEXTSCREEN ;
  OLDTIME := SYSTEM(TIME);
  FOR I := 1 TO 100 DO
    BEGIN
      SEARCHMENU ;
      CLEAR ;
      DISPLAY_SELECTED_TEXTSCREEN;
    END;
  NEWTIME := SYSTEM(TIME) - OLDTIME ;
  SEARCH := ( NEWTIME - OLDTIME ) / 100 - TEXT_TIME ;
  RETURN(SEARCH);
END.
```

Here SEARCHMENU is obtained empirically by having a user make a selection from the appropriate menus. We expect that the running time of the two programs given above will be very close. Any large difference indicates that the effective time used by the selection process is large. (We are assuming that the looping itself takes negligible time. Obvious modifications can be made if this is not the case.)

```

PROGRAM TREE_SEARCH_REPLACE_TEXT ;
(* TIME FOR UPDATING TEXT PAGE WITH SELECTING DEVICE AND
SEARCHING THE TREE OF MENUS *)
BEGIN
TEXTSCREEN;
HEIGHT := 10 ;
OLDTIME := SYSTEM(TIME) ;
FOR I := 1 TO 100 DO
  FOR J := 1 TO HEIGHT DO
    BEGIN
      SEARCHMENU;
      CLEAR;
      DISPLAY_SELECTED_TEXTSCREEN;
      CHANGE_LEVEL;
    END;
  NEWTIME := SYSTEM(TIME);
  LEVEL := ( NEWTIME - OLDTIME ) / (100 * HEIGHT )
    - ( SEARCH + TEXT_TIME ) / HEIGHT ;
  RETURN(LEVEL);
END.

```

A call to the procedure CHANGE_LEVEL involves the changing of menus to a menu for a new level of the menu tree.

The next set of three programs determines a typical time for updating graphics displays. These programs are obtained from the first set by replacing "TEXTSCREEN" by "GRAPHICS_SCREEN"; the details will be omitted.

The remaining two sets of three benchmark programs will measure the efficiency of interfaces in a window environment. Basic concerns here are measurement of the behavior of the window updating system and of the selection devices when the system is under heavy load. Concerns about stationary, pop-up, and pull-down menus; screen layout; and shortcuts through the tree structure hierarchy to frequently used menu items will be postponed to Section 4.

As before, we will present the programs for the updating of text windows and merely indicate the modification necessary for having graphics displays in windows.

```

PROGRAM REPLACE_WINDOW_TEXT;
(* UPDATE TEXT DISPLAYS IN VARIOUS WINDOWS *)
BEGIN
GET_WINDOW_ENVIRONMENT;
SELECT_WINDOW;
CLEAR;
TEXTSCREEN;
OLDTIME := SYSTEM(TIME);
FOR I := 1 TO 100 DO
  BEGIN
    SELECT_WINDOW;
    CLEAR;
    TEXTSCREEN;
  END;
  NEWTIME := SYSTEM(TIME);
  WINDOW_TEXT_TIME := ( NEWTIME - OLDTIME ) / 100 ;
  RETURN (WINDOW_TEXT_TIME);
END.

```

```

PROGRAM SEARCH_REPLACE_WINDOW_TEXT;
(* USE SELECTION DEVICE; REPLACE TEXT IN WINDOW *)
BEGIN
GET_WINDOW_ENVIRONMENT ;
SELECT_WINDOW ;
CLEAR ;
TEXTSCREEN ;
OLDTIME := SYSTEM(TIME);
FOR I := 1 TO 100 DO

```

```

BEGIN
  SEARCH ;
  SELECT_WINDOW ;
  CLEAR ;
  TEXTSCREEN ;
  END ;
  NEWTIME := SYSTEM (TIME) ;
  WINDOW_SEARCH := (NEWTIME - OLDTIME) / 100 - WINDOW_TEXT_TIME ;
  RETURN (WINDOW_SEARCH) ;
  END.

PROGRAM TREE_SEARCH_REPLACE_WINDOW_TEXT ;
(* CHANGE LEVELS OF MENU SELECTION TREE ; SELECT MENU ITEM ;
  REPLACE TEXT SCREEN IN A WINDOW ENVIRONMENT *)
BEGIN
  GET_WINDOW_ENVIRONMENT ;
  SELECT_WINDOW ;
  CLEAR ;
  TEXTSCREEN ;
  HEIGHT := 10 ;
  OLDTIME := SYSTEM(TIME);
  FOR I := 1 TO 100 DO
    FOR J := 1 TO HEIGHT DO
      BEGIN
        SEARCH ;
        SELECT_WINDOW ;
        CLEAR ;
        TEXTSCREEN ;
        CHANGE_LEVEL ;
      END ;
    NEWTIME := SYSTEM(TIME);
    WINDOW_LEVEL := ( NEWTIME - OLDTIME ) / ( 100 * HEIGHT )
      - ( WINDOW_SEARCH + WINDOW_TEXT_TIME ) / HEIGHT;
  RETURN (WINDOW_LEVEL) ;
  END.

```

The remaining three programs are obtained by replacing "TEXTSCREEN" by "GRAPHICS_SCREEN." These programs suggest how a set of benchmarks can be written for the evaluation of the real-time performance of an interface. As such, our viewpoint is different from that of [5] where the main concern is the evaluation of overall window updating speed, rather than the selection process.

These benchmark programs produce the values of SEARCH and LEVEL as described above. They also give the values of WINDOW_SEARCH and WINDOW_LEVEL, which describe the same quantities in a window environment. We will use these values in Section 4.

4. EVALUATION OF AN INTERFACE: ANALYSIS

We now use the quantities SEARCH, LEVEL, WINDOW_SEARCH, and WINDOW_LEVEL to analyze the performance of interfaces for monitoring of processes and for testing and control. Since we are considering only the case of interfaces in a real-time environment, we assume the existence of an external quantity called ABSOLUTE_TIME_LIMIT which represents the maximum time that is available for all of the computer operations including the action of the interface and the retrieval and display of data. Since

the design of the interface is logically independent of the design of the data storage and retrieval algorithms, we restrict our attention to a constant MAX_TIME which is the maximum time that any response of the interface can have. Clearly,

$$\text{MAX_TIME} + \text{RETRIEVAL_TIME} = \text{ABSOLUTE_TIME_LIMIT}.$$

If

$$\text{RETRIEVAL_TIME} \geq \text{ABSOLUTE_TIME_LIMIT},$$

then no interface is possible that meets the required conditions and the hardware and software requirements for the project must be changed. From this point on, we assume that *some* interface is possible within the constraint of MAX_TIME. We also assume:

1. Some portion of the interface requires graphics.
2. The chosen display hardware and software permit mixed text and graphics with a variety of input devices.

3. That this mixture is available both within a window management system and outside it.

4. $SEARCH < WINDOW_SEARCH$.

5. $LEVEL < WINDOW_LEVEL$.

Assume that there is a total of M options available from menus and that the menus are arranged hierarchically. Then the two extremes of each node of the tree (except the last) having one child and the root having $M - 1$ children correspond to times of $M \cdot LEVEL$ and $M \cdot SEARCH$, respectively. If each of these numbers is smaller than MAX_TIME , then every arrangement of menus is feasible from a real-time performance viewpoint and only human factors considerations need be considered. A similar statement holds for $WINDOW_LEVEL$ and $WINDOW_SEARCH$. We thus restrict our attention to the case that at least one of

$$M \cdot LEVEL > MAX_TIME, \quad (1)$$

$$M \cdot SEARCH > MAX_TIME, \quad (2)$$

$$M \cdot WINDOW_LEVEL > MAX_TIME, \quad (3)$$

$$M \cdot WINDOW_SEARCH > MAX_TIME \quad (4)$$

is true. Note that we make no assumption about which of $LEVEL$ and $SEARCH$ or which of $WINDOW_LEVEL$ and $WINDOW_SEARCH$ is larger.

There are several alternatives available to improve the real-time performance of the interface.

1. If eqn (1) holds and eqn (2) does not, then increasing the degree of each node at the cost of increasing $SEARCH$ will speed up the worst case performance.

2. If eqn (2) holds but eqn (1) does not, then decreasing the degree of each node at the cost of increasing $LEVEL$ will improve worst case performance.

3. If eqn (1) and eqn (2) both hold, then the only inexpensive solution is to use windowing. Clearly eqn (3) and eqn (4) are also true as stated. However, in this case we note that many different choices may be displayed in windows and thus (1) and (2) should be replaced by

$$M \cdot WINDOW_LEVEL /$$

$$NUMBER_OF_WINDOWS < MAX_TIME, \quad (3a)$$

$$M \cdot WINDOW_SEARCH /$$

$$NUMBER_OF_WINDOWS < MAX_TIME. \quad (4a)$$

We apply the same analyses used in cases 1 and 2 in this situation.

4. In each of cases 1-3, we assumed that there were no shortcuts available in the hierarchical menu structure. However, many selection devices (such as a three button mouse) allow for omitting several intermediate levels of a tree. This should be used for experienced users to increase operator satisfaction and to increase speed of the interface. This was pointed out in [1]. We

also suggest that only stationary or pull-down menus be used for clarity.

5. Not all operations in the monitoring of processes or testing and control need be carried out in real-time. Those operations should be on the bottom level of the tree so as to allow faster traversals of the tree in most cases.

6. Use an additional human operator to reduce the load on the interface by a factor of 2.

7. Revise the underlying hardware and software.

These guidelines should not be considered complete but instead as suggestions for actions when an interface for real-time activity is predicted to be overloaded.

5. SUMMARY

In this note, we have described an example of a user interface for monitoring of processes; testing and control; and display of three dimensional motion and how it evolved over time. The problems encountered there were abstracted and generalized in order to develop a general model for the analysis of the performance of human computer interfaces for programs which are used for monitoring of processes and for testing and control. Emphasis was given to specific benchmarks and guidelines for the design and evaluation of "real-time" interfaces, rather than the general performance measurements often used [5].

REFERENCES

1. J. D. Foley, V. L. Wallace and P. Chan, The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications* 4, 13-48 (1984).
2. I. Carlsson and J. Michener, Quantitative analysis of vector graphics performance. *ACM Trans. Graphics* 2, 57-88 (1982).
3. J. H. Clark, The geometry engine, a VLSI system for graphics. *Computer Graphics (ACM)* 16, 127-134 (1982).
4. R. Leach, Graphical control systems and multiple displays. *Comput. & Graphics* 9, 415-422 (1986).
5. R. Zabih and R. Jain, A performance comparison of the window systems of two LISP machines. *Proceedings of the Fourteenth Annual Computer Science Conference*, p. 458. Cincinnati, Ohio (Feb. 1986).
6. C. M. Mitchell, L. J. Stewart, A. K. Bocast and E. D. Murphy, Human Factors Aspects of Control Room Design: Guidelines and Annotated Bibliography. NASA Technical Memorandum 84942 (1982).
7. C. M. Mitchell, P. Van Balen and K. Moe (Ed.), *Proceedings of the Human Factors Conference in System Design Symposium*. NASA/GSFC Human Factors Group, Greenbelt, MD. (May 1982).
8. NASA, DOCS Delta Critical Design Review. NASA Technical Memorandum, (Jan., 1984).
9. D. Mandl, D. Carlton and B. Buchanan, COBE Design Review. NASA Technical Memorandum (Dec. 1984).
10. Microtouch Screen Specifications, Microtouch Systems, Inc, Woburn, MA (1985).
11. A. Albert, The effect of graphics input devices on performance in cursor positioning tasks. *Proc. Human Factors Conference* (1982).
12. S. Card, User perceptual mechanisms in the search of computer command menus. *Proc. Human Factors in Computer Systems Conference*, pp. 190-196. Washington, D.C. (1982).
13. J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA (1982).